universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Exploring ActivityPub Interoperability for Bluesky

verfasst von | submitted by

Martin Sonnberger BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 935
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Math. Dr. Peter Reichl Privatdoz.

Acknowledgements

Thank you to my supervisors, DI Paul Fuxjäger and Prof. Peter Reichl, for their guidance, support, and encouragement throughout my research journey.

Thank you to my parents for their never-ending support during my studies.

Abstract

The emergence of decentralized social media has produced two major, mutually incompatible platforms: Mastodon and Bluesky. While both aim to return ownership and control to users—Mastodon through the ActivityPub protocol and Bluesky through the AT Protocol—the lack of interoperability between them forces users to choose one community over the other, fragmenting the decentralized social web.

This thesis presents *Fedisky*, a sidecar service that enables federation between Bluesky and Mastodon. By running alongside an existing Bluesky Personal Data Server (PDS), Fedisky exposes ActivityPub-compatible endpoints that allow Mastodon users to discover, follow, and interact with Bluesky users without needing to create accounts on Bluesky. Posts flow to Mastodon followers in real time, while replies and likes from Mastodon are relayed back to Bluesky in a transparent and attributable way.

Rather than routing traffic through a centralized bridge, each PDS operator independently deploys their own instance, preserving autonomy and avoiding single points of failure. This thesis evaluates this approach against centralized bridging services and PDS forks, finding that the sidecar pattern offers the best balance of identity cohesion, operator autonomy, and long-term maintainability, while acknowledging inherent asymmetries in how authorship and engagement are represented across the protocol boundary.

Kurzfassung

Das Aufkommen dezentraler sozialer Medien hat zwei große, miteinander inkompatible Plattformen hervorgebracht: Mastodon und Bluesky. Während beide darauf abzielen, Eigentumsrechte und Kontrolle an die Nutzer zurückzugeben – Mastodon über das ActivityPub-Protokoll und Bluesky über das AT-Protokoll – zwingt die mangelnde Interoperabilität zwischen ihnen die Nutzer dazu, sich für eine der beiden Communities zu entscheiden, was zu einer Fragmentierung des dezentralen sozialen Webs führt.

Diese Arbeit stellt *Fedisky* vor, einen Sidecar-Dienst, der eine Föderation zwischen Bluesky und Mastodon ermöglicht. Durch den Betrieb parallel zu einem bestehenden Bluesky Personal Data Server (PDS) stellt Fedisky ActivityPub-kompatible Endpunkte bereit, die es Mastodon-Nutzern ermöglichen, Bluesky-Nutzer zu entdecken, ihnen zu folgen und mit ihnen zu interagieren, ohne ein Konto auf Bluesky erstellen zu müssen. Beiträge werden in Echtzeit an Mastodon-Follower weitergeleitet, während Antworten und Likes von Mastodon auf transparente und zurückverfolgbare Weise an Bluesky zurückgesendet werden.

Anstatt den Datenverkehr über eine zentralisierte Brücke zu leiten, stellt jeder PDS-Betreiber unabhängig seine eigene Instanz bereit, wodurch die Autonomie gewahrt und einzelne Ausfallpunkte vermieden werden. Diese Arbeit bewertet diesen Ansatz im Vergleich zu zentralisierten Brückendiensten und PDS-Forks und kommt zu dem Ergebnis, dass das Sidecar-Muster die beste Balance zwischen Identitätskohäsion, Betreiberautonomie und langfristiger Wartbarkeit bietet, wobei inhärente Asymmetrien in der Darstellung von Urheberschaft und Interaktion über die Protokollgrenze hinweg anerkannt werden.

Contents

List of Tables

List of Figures

Listings

1 Introduction

1.1 Motivation

For much of the 2010s, microblogging was dominated by centralized platforms, most notably X (formerly known as Twitter). In 2022, the acquisition of Twitter by Elon Musk prompted widespread concern over platform governance, content moderation policy, and API access, triggering a large-scale migration of users to decentralized alternatives [25, 28]. Mastodon, a federated microblogging platform built on the ActivityPub protocol [40], saw its monthly active user count grow from approximately 300 000 to 1.8 million within months of the acquisition [25]. Bluesky, built on the AT Protocol [27], opened public registration in February 2024 and grew from 3 million to over 25 million users by the end of that year [9]. A second significant wave of departures from X followed the 2024 US presidential election, driven by the concerns over the platform's political alignment under Musk. Bluesky's user base doubled in the 90 days surrounding the election, gaining over one million new sign-ups in a single week [24].

Both platforms share the goal of decentralizing social media: rather than relying on a single operator, they allow independent servers to participate in a federated network. However, they achieve this through fundamentally different protocols. Mastodon and the broader Fediverse use ActivityPub, a W3C Recommendation that defines a push-based federation model. Bluesky uses the AT Protocol, which takes a pull-based approach built around signed data repositories and global indexing [27]. As a result, users on one network cannot interact with users on the other, fragmenting the decentralized social media landscape into two incompatible ecosystems.

This fragmentation undermines the very premise of decentralization. If the goal is to free users from dependence on a single platform, locking them into one of two competing protocol ecosystems only shifts the problem rather than solving it. Bridging these ecosystems would allow users to remain on their preferred platform while still reaching people on the other side, moving closer to an interconnected open social web.

Existing efforts to bridge ActivityPub and AT Protocol, most notably Bridgy Fed [7], operate as centralized services. A single bridge instance mediates all cross-protocol traffic, creating a single point of failure and removing control from individual server operators. This thesis explores whether a decentralized, per-instance approach to bridging is feasible; one where each AT Protocol server operator can independently choose to enable federation with the Fediverse.

1.2 Problem Statement

AT Protocol and ActivityPub differ in nearly every dimension relevant to federation. Their identity models are incompatible: AT Protocol identifies users by Decentralized Identifiers (DIDs) and resolves human-readable handles through DNS, while ActivityPub identifies actors by HTTPS URIs discovered via the WebFinger protocol. Their content representations diverge: AT Protocol stores posts as plain text with byte-range annotations (facets) for links and mentions, while ActivityPub transmits content as HTML. Their delivery mechanisms are fundamentally different: AT Protocol uses a pull-based model where consumers subscribe to a firehose of repository commits, while ActivityPub uses a push-based model where the origin server delivers signed activities to each follower's inbox. Finally, their engagement semantics differ: AT Protocol records likes and reposts as repository records, while ActivityPub delivers them as transient activities.

No standardization mechanism exists for interoperability between the two protocols. Existing bridging solutions operate as centralized services, creating single points of failure and removing operator control over the bridging process. The question is whether a decentralized, per-instance approach to bridging is feasible, and if so, what technical challenges and architectural trade-offs it entails.

1.3 Research Questions

This thesis addresses the following research questions:

RQ1 How can a sidecar service enable ActivityPub federation for an AT Protocol Personal Data Server?

RQ2 What are the key technical challenges and trade-offs when bridging AT Protocol and ActivityPub?

RQ3 How does a sidecar architecture compare to alternative approaches for cross-protocol social media federation?

RQ1 is addressed through the design and implementation of Fedisky, a working sidecar service that bridges AT Protocol and ActivityPub. RQ2 is answered by documenting the technical challenges encountered during implementation and the trade-offs made in resolving them. RQ3 is addressed through a qualitative comparison of the sidecar architecture with a centralized bridging service and a Personal Data Server (PDS) fork, evaluating along four dimensions: user experience, scalability, maintainability, and modularity.

1.4 Methodology

This thesis follows a constructive research approach, where the primary contribution is the design and implementation of a software artifact that addresses the stated research questions. The methodology consists of three phases:

First, we *design and implement* the sidecar service, making architectural decisions informed by the requirements of both protocols and the constraints of the sidecar pattern. The DID serves as the canonical identity anchor: each Bluesky user's DID is used to derive their ActivityPub actor URI, establishing a unidirectional identity projection from AT Protocol into the Fediverse.

Second, we *document the technical challenges* encountered during implementation, including identity mapping, content format conversion, delivery method bridging, and engagement notification handling. For each challenge, we describe the approach taken and the trade-offs involved.

Third, we *compare the sidecar architecture* with two alternatives—a centralized bridging service (as realized by Bridgy Fed) and a PDS fork—along the dimensions of user experience, scalability, maintainability, and modularity. This comparison is qualitative, drawing on our implementation experience and analysis of the alternative approaches.

1.5 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 provides background on DIDs, the AT Protocol, and ActivityPub, and discusses related work including Bridgy Fed and Wafrn. Chapter 3 presents the design and implementation of Fedisky, covering its architecture, core flows, conversion layer, and operational considerations. Chapter 4 discusses the sidecar architecture in comparison to alternative approaches and examines the challenges and trade-offs encountered during protocol bridging. Chapter 5 concludes the thesis with a summary and directions for future work.

2 Background and Related Work

2.1 Decentralized Identifiers (DIDs)

A Decentralized Identifier (DID) is a globally unique, persistent identifier that does not require a centralized registration authority [39]. Unlike traditional identifiers such as email addresses or domain names, which depend on centralized registries, DIDs are designed to be under the control of the entity they identify. The W3C published the DID specification as a Recommendation in July 2022 [39], with version 1.1 currently in Candidate Recommendation status [38].

2.1.1 Syntax and Resolution

Every DID conforms to a URI scheme composed of three parts: the fixed `did:` prefix, a method name, and a method-specific identifier [39]. For example, `did:plc:d7mawpfc` uses the method `plc` with the method-specific identifier `d7mawpfc`. The method name determines how the DID is created, resolved, updated, and deactivated [39].

Resolving a DID produces a *DID document*, a data structure that describes the DID subject and provides the information necessary to interact with it cryptographically. Listing 2.1 shows the general structure of a DID document.

```
{
  "@context": ["https://www.w3.org/ns/did/v1"],
  "id": "did:example:123456789abcdefghi",
  "controller": "did:example:123456789abcdefghi",
  "verificationMethod": [{
    "id": "did:example:123456789abcdefghi#keys-1",
    "type": "Ed25519VerificationKey2020",
    "controller": "did:example:123456789abcdefghi",
    "publicKeyMultibase": "zH3C2AVvLMv6gmMNam..."
  }],
  "authentication": [
    "did:example:123456789abcdefghi#keys-1"
  ],
  "service": [{
    "id": "did:example:123456789abcdefghi#service-1",
    "type": "ExampleService",
    "serviceEndpoint": "https://example.com/service"
  }]
}
```

Listing 2.1: Structure of a DID document (adapted from [39])

The key properties of a DID document are [39]:

Verification methods Cryptographic public keys that enable the DID subject or its delegates to prove control over the identifier, for instance to authenticate or to produce digital signatures.

Verification relationships Declarations of the purposes for which each verification method may be used, such as authentication, assertion, or key agreement. In Listing 2.1, the `authentication` array references a verification method by its identifier, indicating that this key may be used for authentication.

Service endpoints URIs through which the DID subject can be contacted or through which services associated with the subject can be discovered.

Controllers One or more DIDs authorized to make changes to the DID document.

2.1.2 DID Methods

The DID specification is deliberately agnostic about the underlying infrastructure used to manage identifiers. The concrete mechanisms for creating, resolving, updating, and deactivating DIDs are defined by individual *DID methods* [39]. Each method specifies its own resolution protocol and may rely on different trust models. Two methods are particularly relevant to this thesis:

`did:plc`. The `did:plc` method was developed by Bluesky specifically for the AT Protocol [13]. A `did:plc` identifier is derived from the cryptographic hash of its initial creation operation, making it self-authenticating. Control over the identifier resides in a set of *rotation keys*, ordered by authority. Each update to the identity—such as key rotation or a change of hosting provider—is recorded as a signed, immutable operation that references its predecessor by hash, forming a verifiable chain [13]. A centralized Public Ledger of Credentials (PLC) directory server validates and orders these operations, maintaining a publicly accessible audit log. While the directory represents a single point of trust, its potential for abuse is limited: the operation log is self-certifying, meaning that any observer can independently verify the current state of a DID document from the signed operation history [13]. Higher-authority rotation keys can invalidate operations signed by lower-authority keys within a 72-hour recovery window, enabling account recovery in the event of a key compromise [13].

In practice, the PLC directory is currently operated by Bluesky Social PBC. Acknowledging the centralization that this entails, Bluesky announced in September 2025 the formation of an independent Swiss Association to assume operation of the directory, and explicitly stated that a single global directory is not expected to remain the long-term technical architecture [10]. The longer-term governance and topology of the system are therefore still open, with transparency logs and federated mirrors among the directions and active discussions [18].

The specification does not prescribe where rotation keys are held. In the prevailing deployment pattern, the PDS operator generates and retains the highest-priority rotation key on the user's behalf during account creation. Users may enroll an additional

self-controlled rotation key of higher priority, which prevents the PDS operator from unilaterally re-keying the account or migrating it to a different host [1].

`did:web`. The `did:web` method uses the DNS and HTTPS to host DID documents at well-known URLs [33]. A `did:web` identifier maps directly to an HTTPS path: for example, `did:web:example.com` resolves to `https://example.com/.well-known/did.json`, while `did:web:example.com:user:alice` resolves to `https://example.com/user/alice/did.json` [33]. Because it builds on existing web infrastructure, `did:web` is straightforward to deploy but inherits the trust assumptions of DNS and Transport Layer Security (TLS) certificate authorities. Updates are performed by modifying the hosted document directly; deactivation is achieved by removing it [33].

2.1.3 DIDs as a Cross-Protocol Identity Primitive

The design goals of the DID specification include decentralization, cryptographic verifiability, and interoperability [39]. Crucially, a DID is not bound to any particular application protocol. An AT Protocol user is identified by a DID, but that same DID can in principle serve as an identity anchor in other contexts.

This protocol independence makes DIDs a natural foundation for bridging. In this thesis, Fedisky uses the Bluesky user's `did:plc` identifier as the canonical identity from which the corresponding ActivityPub actor URI is deterministically derived. The DID therefore functions as a shared reference point across the two protocols, enabling a stable identity mapping without requiring users to register separate accounts on each side of the bridge. The details of this identity projection are described in Chapter 3.

2.2 AT Protocol

The AT Protocol (Authenticated Transfer Protocol, or atproto) is a decentralized foundation for public social media, developed by Bluesky Social PBC [27]. Its design is informed by four goals: enabling decentralization through interoperable providers for every part of the system, making it easy for users to switch between providers, giving users agency over the content they see, and providing a user experience comparable to centralized services [27]. The protocol separates infrastructure hosting from content moderation and algorithmic curation, allowing different sets of people to offer these services independently. This separation is motivated by the observation that the skills needed to run infrastructure differ substantially from those required for community moderation [27].

An IETF Internet-Draft formalizing the protocol architecture was submitted in September 2025 [32]. The following subsections introduce the protocol components that are relevant to the implementation described in Chapter 3.

2.2.1 Architecture Overview

The AT Protocol network consists of several interacting service roles, each of which can be operated independently by different providers [32]. Figure 2.1 illustrates the relationships

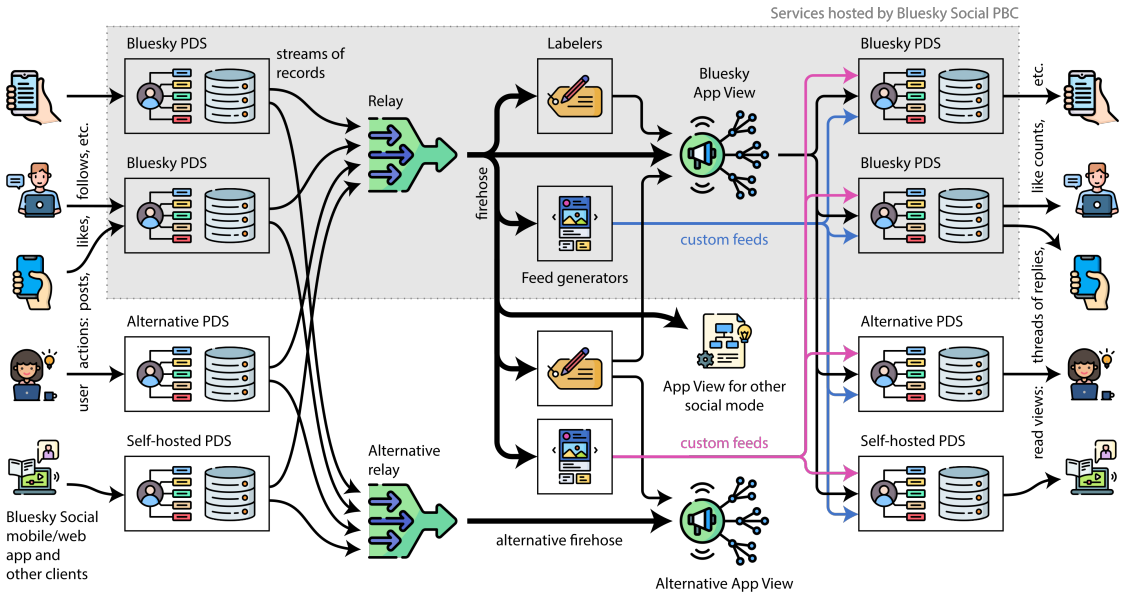

Figure 2.1: Architecture of the AT Protocol network, showing the main service roles and data flows between them (reproduced from [27]).

and data flows between these components.

Personal Data Server (PDS). A PDS hosts one or more user accounts and their data repositories [27]. It stores each user's repository and signing keys, serves the repository data via an HTTP API, and provides a real-time stream of repository updates over a WebSocket [32]. Hosting a PDS for a small number of users requires minimal computing resources, making self-hosting feasible even on low-cost hardware [27]. Crucially, the PDS endpoint is an implementation detail not exposed to end users; from the perspective of user interaction, whether two accounts share a PDS is irrelevant [27]. Migration between PDS providers consists of transferring the repository and associated blobs to the new host and issuing a DID document update to point to the new PDS endpoint [27].

Relay. A relay crawls repositories from all known PDS instances, verifies signatures and Merkle tree proofs, and produces the *firehose*: an aggregated stream of repository updates from across the network [27]. The firehose is publicly available and provides a convenient entry point for building network-wide indexes, since the relay pre-filters the aggregated stream, rejecting malformed records and high-volume spam before subscribers see them. However, it does not otherwise interpret or index the records it forwards [27].

AppView. An AppView consumes the firehose and processes the records relevant to a particular application. For the Bluesky microblogging application, the AppView counts likes, aggregates reply threads, maintains follower sets, and constructs user timelines [27]. It exposes this indexed data via an HTTP API [32]. The AppView is application-specific:

a different social media application built on top of the AT Protocol would require its own AppView to index the relevant record types [27].

Labelers and Feed Generators. The AT Protocol separates "opinionated" services—those that make subjective judgments about content—from the neutral indexing infrastructure [27]. *Labelers* consume the firehose and produce a stream of labels (e.g. flagging a post as suspected spam), which AppViews and clients use to apply content filtering. *Feed generators* return ordered lists of post identifiers selected according to custom criteria, enabling algorithmic feeds that users can subscribe to [27]. Both services can be operated by anyone, enabling an open market of moderation opinions, where competing services may reach different conclusions about the same content [27].

2.2.2 Identity

AT Protocol identity builds on the DID infrastructure described in Section 2.1. Every account is identified by a DID, which serves as the stable, canonical identifier [12]. Two DID methods are supported: `did:plc` and `did:web` (introduced in Section 2.1.2), the latter restricted to hostname-level identifiers without paths [12].

The protocol requires that every resolved DID document contains three AT-Protocol-specific entries [12]:

1. A *signing key* (a verification method with the fragment `#atproto`) used to verify the cryptographic signatures on repository commits.

2. A *PDS service endpoint* (a service entry with the fragment `#atproto_pds`) pointing to the HTTPS URL of the user's Personal Data Server.

3. A *handle claim* in the `alsoKnownAs` array, encoded as an `at://` URI followed by the handle hostname.

Handles. Handles are human-readable usernames that take the form of DNS domain names, such as `alice.bsky.social` [15]. A handle must maintain a bidirectional link with its corresponding DID: the handle must resolve to the DID, and the DID document must claim the handle [27, 15]. The forward link (handle to DID) is established either via a DNS TXT record at `_atproto.handle` or via an HTTPS endpoint at `/.well-known/atproto-did` on the handle's domain [15]. The reverse link (DID to handle) is the `alsoKnownAs` entry in the DID document [15].

Using DNS domain names as handles allows users to change their handle without affecting their social graph, since all inter-user references in the protocol use DIDs rather than handles [27]. It also means that organizations with well-known domain names can use those names directly—for example, the New York Times uses the handle `nytimes.com`—and can grant staff subdomains to indicate affiliation [27].

2.2.3 Data Model

Repositories. Each AT Protocol account has exactly one *repository*, a self-certifying data store containing all of the user's public records [11]. Every user action—posting, liking, following—results in a typed record appended to the user's own repository [27]. Each user writes exclusively to their own store. A follow, for instance, produces a record solely in the follower's repository. Consequently, computing the full follower set of an account demands a network-wide index rather than a single lookup [27].

Records within a repository are organized into *collections*, identified by Namespace Identifiers (NSIDs) [32]. Within a collection, each record is addressed by a *record key*. The combination forms a repository-internal path of the form `<collection>/<record-key>`; for example, `app.bsky.feed.post/123456abcdef`. Records are globally addressable via AT URIs: `at://<did>/<collection>/<record-key>` [32].

Merkle Search Tree. The records in a repository are stored in a Merkle Search Tree (MST) [3], a deterministic, content-addressed search tree that combines three properties: a given set of items has a unique representation as a tree, key order is preserved, and the tree remains balanced probabilistically [3]. The structure is similar to a B-tree in that internal nodes contain multiple values that partition the key space for their children, but the shape of the tree is determined by hashing the keys rather than by insertion order. Each key is assigned to a layer based on the number of leading zeros in its hash; keys with more leading zeros sit higher in the tree, yielding a probabilistically balanced structure with an expected branching factor determined by the hash function's base [3]. Since the tree is deterministic, two repositories containing the same records will always produce the same root hash, regardless of the order in which the records were inserted. After every change to a repository, the root hash of the MST is included in a signed *commit object* [11]. The signing key used to produce this signature is published in the user's DID document, enabling anyone to verify that a given record authentically belongs to a given user's repository [27]. This property—that repositories are cryptographically authenticated—is the origin of the protocol's name: *Authenticated Transfer* [27]. The deterministic structure also makes synchronization efficient: when comparing two versions of a repository, branches with identical hashes can be skipped entirely, so only the differing records need to be transferred [3].

Serialization. Records are encoded in DAG-CBOR, a restricted form of Concise Binary Object Representation (CBOR) [32]. JSON is used in web APIs and for human consumption, while deterministic CBOR is used for hashing, signatures, and storage [32]. Media files such as images and videos are not stored within the repository itself but as separate blobs, referenced from records by their content hash [27].

2.2.4 Lexicons

A *Lexicon* is a schema definition language used to describe AT Protocol records, HTTP endpoints, and event stream messages [17]. Lexicons serve a role analogous to JSON

Schema or OpenAPI, but include a global namespace system based on reverse-domain-name NSIDs [17]. For example, the NSID `app.bsky.feed.post` defines the schema for Bluesky microblog posts, while `com.atproto.sync.subscribeRepos` defines the firehose event stream.

Lexicons enable multiple social modes to coexist on the same infrastructure [27]. The `com.atproto` namespace defines core protocol concepts such as identity and repository synchronization, while the `app.bsky` namespace defines the Bluesky microblogging application. Anyone can define a new lexicon with new record types, and these records can be stored without requiring changes to PDSes or relays [27].

2.2.5 Content Representation

In the `app.bsky` lexicon, a post record (`app.bsky.feed.post`) contains the post's text as a plain UTF-8 string of up to 300 graphemes, along with optional metadata such as creation timestamp, language tags, and reply references [21].

Facets. Rich-text annotations are represented through *facets*: byte-range indexed annotations attached to the post text [21]. Each facet specifies a byte range within the text and one or more features, such as a mention (referencing another user's DID), a hyperlink (referencing an external URI), or a hashtag [21]. This approach separates the plain-text content from its annotations, in contrast to ActivityPub's use of inline HTML markup (Section 2.3).

Embeds. Media and references to other content are attached as *embeds* rather than being inline in the text [21]. The `app.bsky` lexicon defines embed types for images (up to four per post, each with required alt text) [19], video (a single video blob with optional captions) [20], external links (with title, description, and optional thumbnail), and quote posts (referencing another post by its AT URI) [21].

2.2.6 Event Stream (Firehose)

The AT Protocol's real-time synchronization mechanism is the *event stream*, commonly referred to as the *firehose* [14]. Both PDSes and relays expose event streams via WebSocket endpoints, with messages encoded as concatenated CBOR objects: a header specifying the message type and a payload containing the event data [14].

The primary event stream endpoint is `com.atproto.sync.subscribeRepos`, which emits events whenever records are created, updated, or deleted in the repositories hosted by the server [32]. Each event carries a monotonically increasing sequence number that enables cursor-based resumption: a consumer that disconnects can reconnect with the last successfully processed sequence number and receive any events it missed, provided they fall within the server's backfill window [14].

Repository mutations are transmitted as partial Content Addressable aRchive (CAR) files containing only the changed blocks [11]. A consumer of the firehose receives commit-level events that include the account DID, the revision identifier, and the set of record

operations (creates, updates, deletes) within that commit [32]. This is the mechanism through which Fedisky learns about new posts and other actions on the PDS it is bridging, as described in Chapter 3.

2.3 ActivityPub

ActivityPub is a W3C Recommendation published in January 2018 that defines a decentralized social networking protocol [40]. It builds on the Activity Streams 2.0 data model [35, 36] and defines two protocol layers: a *client-to-server* (C2S) layer through which clients submit activities to a server, and a *server-to-server* (S2S) layer through which servers federate activities across domain boundaries [40]. An implementation may support either layer or both; Fedisky implements only the S2S layer, since it acts as a federating server rather than hosting end-user clients.

ActivityPub is the protocol underlying a network of independently operated servers collectively known as the *Fediverse*. The most widely deployed ActivityPub implementation is Mastodon, a microblogging platform, but many other applications—including Pixelfed (image sharing), PeerTube (video hosting), and Lemmy (link aggregation)—also federate using ActivityPub. The diversity of implementations means that the protocol must be understood at two levels: the W3C specification itself and the de facto conventions established by dominant implementations.

2.3.1 Data Model: Activity Streams 2.0

ActivityPub's data model is defined by the Activity Streams 2.0 specification, which consists of a core document describing the abstract model and serialization [35] and a vocabulary document defining concrete types [36]. The model is built on three fundamental concepts [35]:

Objects represent entities such as a text note, an image, or a person. Every object may carry properties including `id` (a globally unique URI), `type`, `content`, `published`, `attributedTo`, and `inReplyTo` [36].

Activities are a subtype of Object that describe actions. An activity typically has an `actor` (who performed the action), an `object` (what was acted upon), and optionally a `target` [35]. The vocabulary defines activity types such as `Create`, `Delete`, `Update`, `Follow`, `Accept`, `Reject`, `Like`, `Announce` (analogous to repost), and `Undo` [36].

Collections are ordered or unordered sets of objects, used to represent an actor's outbox, inbox, followers list, and similar aggregations. Large collections are paginated using `CollectionPage` objects linked by `next` and `prev` references [35].

Activity Streams documents are serialized as JSON and are designed to be compatible with JSON-LD [37], a format that maps short property names to globally unique IRIs via

an `@context` declaration [35]. In practice, most ActivityPub implementations treat the JSON-LD context as a fixed preamble and process documents as plain JSON rather than performing full linked-data expansion [30].

2.3.2 Actors

The central abstraction in ActivityPub is the *actor*. An actor represents an entity—typically a user account—capable of performing and receiving activities [40]. Each actor is identified by a globally unique HTTPS URI that, when dereferenced, returns a JSON-LD document describing the actor. The specification requires every actor to have an `inbox` and an `outbox`, both of which are `OrderedCollection` endpoints, and recommends `followers` and `following` collections [40]. The vocabulary defines several actor types, of which `Person`, `Application`, and `Service` are most commonly used [36].

Listing 2.2 shows the structure of an ActivityPub actor document as served by Mastodon.

```
{
  "@context": [
    "https://www.w3.org/ns/activitystreams",
    "https://w3id.org/security/v1"
  ],
  "id": "https://mastodon.social/users/alice",
  "type": "Person",
  "preferredUsername": "alice",
  "inbox": "https://mastodon.social/users/alice/inbox",
  "outbox": "https://mastodon.social/users/alice/outbox",
  "followers": "https://mastodon.social/users/alice/followers",
  "following": "https://mastodon.social/users/alice/following",
  "publicKey": {
    "id": "https://mastodon.social/users/alice#main-key",
    "owner": "https://mastodon.social/users/alice",
    "publicKeyPem": "-----BEGIN PUBLIC KEY-----\n..."
  }
}
```

Listing 2.2: Structure of an ActivityPub actor document (simplified).

Although the ActivityPub specification does not mandate a specific authentication mechanism, the Fediverse has converged on *HTTP Signatures*: every actor publishes an RSA or Ed25519 public key in its actor document, and outgoing requests are signed using the corresponding private key [30]. The receiving server verifies the signature by fetching the sender's actor document and checking the signature against the embedded public key. This mechanism serves both authentication (confirming the request originates from the claimed actor) and integrity (ensuring the request body has not been tampered with).

2.3.3 Discovery: WebFinger

ActivityPub actors are identified by HTTPS URIs, but users typically refer to one another using human-readable addresses of the form `@user@domain`. The translation from such

an address to an actor URI is performed using the WebFinger protocol, defined in RFC 7033 [26].

A WebFinger query is an HTTPS GET request to the well-known endpoint `/.well-known/webfinger` on the user's domain, with the account encoded as an `acct:` URI in the `resource` query parameter [26]. For example, looking up `@alice@mastodon.social` produces the request:

```
GET https://mastodon.social/.well-known/webfinger?resource=acct:
                        alice@mastodon.social
```

The server responds with a JSON Resource Descriptor (JRD) containing a `subject` field and a `links` array [26]. The ActivityPub actor URI is found in the link with relation type `self` and media type `application/activity+json` [30]. WebFinger thus serves a similar role to AT Protocol handle resolution (Section 2.2.2): both map a human-readable name to a canonical, machine-usable identifier.

2.3.4 Activities and Delivery

Federation in ActivityPub follows a push-based model. When an actor performs an action, the origin server constructs an activity, places it in the actor's outbox, and delivers it to the inboxes of all intended recipients [40]. Recipients are specified via the `to`, `cc`, `bto`, and `bcc` properties on the activity; the server resolves each addressee to an actor, dereferences the actor to obtain its inbox URL, and POSTs the activity to that inbox [40]. Blind recipients (`bto` and `bcc`) are removed from the activity before delivery [40].

The special collection `https://www.w3.org/ns/activitystreams#Public` is used to indicate that an activity is publicly visible. However, servers do not actually deliver to this URI, it rather serves as a marker that the activity should be accessible without authentication [40].

As an optimization, ActivityPub allows servers to advertise a *shared inbox*: a single endpoint that accepts activities addressed to any actor on that server [40]. When delivering to multiple recipients on the same server, the sending server may POST the activity once to the shared inbox instead of once per recipient.

This push-based delivery model contrasts with the AT Protocol's approach, where a central relay pulls updates from all repositories and produces an aggregated firehose (Section 2.2.6). In ActivityPub, each server is responsible for delivering its own activities to all relevant recipients, and there is no global index of all content.

2.3.5 Content Representation

The most common object type for microblogging content is `Note`, a short-form text object [36]. Unlike AT Protocol posts, which store text as plain UTF-8 with separate facet annotations (Section 2.2.5), ActivityPub notes carry their content as HTML in the `content` property [36]. Mentions, hashtags, and links are represented as inline HTML elements. Mastodon sanitizes incoming HTML to a limited set of safe elements—primarily

paragraph, line break, anchor, and span tags, with later versions adding support for lists, emphasis, and code blocks [30].

Media files such as images, audio, and video are attached to objects via the `attachment` property, each specifying a `url`, `mediaType`, and optional `name` (used as alt text) [36]. Content warnings are conveyed through the `summary` property along with a `sensitive` flag [30].

Mastodon determines visibility of a post from the combination of addressing properties and mention tags: a post addressed `to` the public collection is public, one that places the public collection only in `cc` is unlisted, and a post addressed only to the actor's followers collection is private [30].

The difference in content representation between the two protocols—HTML with inline markup versus plain text with byte-indexed facets—is one of the key conversion challenges addressed in the bridge implementation described in Chapter 3.

2.4 Related Work

2.4.1 Bridgy Fed

Bridgy Fed is a bridge service that connects the Fediverse (ActivityPub), Bluesky (AT Protocol), and the IndieWeb (webmentions and microformats2), providing fully bidirectional translation of follows, posts, replies, likes and reposts between these networks [8]. It is developed and operated by Ryan Barrett, and is now maintained under A New Social, a nonprofit organization [6].

Architecture. Bridgy Fed operates as a single centralized service hosted on Google Cloud Platform [5]. Its architecture consists of three internal services: a *frontend* that handles incoming HTTP requests including WebFinger queries, ActivityPub inbox endpoints, and AT Protocol XRPC methods; a *router* that receives events from all supported protocols and dispatches them to destination protocols via task queues; and a *hub*, a single-instance service that maintains a persistent WebSocket connection to the Bluesky relay firehose [5]. When the hub receives an event from the firehose that involves a bridged user, it enqueues a task for the router, which converts the content into the destination protocol's format and delivers it [5].

The service abstracts across protocols by converting all content into a common internal format (Activity Streams 1) and only translating it to the destination protocol's native representation at delivery time [5]. Format conversion is handled by a separate library (*granary*), while AT Protocol repository operations are delegated to another library (*arroba*) [5].

Identity. Bridgy Fed creates proxy identities for bridged users on the destination network. A Fediverse user such as `@alice@mastodon.social` appears on Bluesky under the handle `alice.mastodon.social.ap.brid.gy`, while a Bluesky user such as `alice.bsky.social` appears in the Fediverse as `@alice.bsky.social@bsky.brid.gy` [8]. All bridged identities

thus share the `brid.gy` domain, making them visually distinguishable from native accounts but also tying every bridged identity to Bridgy Fed's infrastructure.

Consent model. Bridging is opt-in: users must explicitly enable it, either by logging in to the Bridgy Fed web interface or by following a designated bot account on their network [8]. Only full public posts are bridged; unlisted, followers-only, and private posts are excluded [8]. Users can opt out at any time by blocking the bot account [8]. The opt-in model was a deliberate response to concerns from Fediverse communities about content being shared across network boundaries without user consent [7].

Comparison with Fedisky. Although Bridgy Fed and Fedisky address the same fundamental problem of enabling interaction between AT Protocol and ActivityPub, they differ significantly in architecture (centralized service vs. per-instance sidecar), identity model (shared `brid.gy` domain vs. operator-controlled domain), and scope (three protocol families vs. AT Protocol-to-ActivityPub only). A detailed comparison is presented in Chapter 4.

2.4.2 Wafrn

Wafrn is an open-source, Tumblr-inspired social networking platform that natively implements both ActivityPub and the AT Protocol within a single application [2]. Unlike Bridgy Fed and Fedisky, which act as intermediaries between two separate protocol networks, Wafrn is itself a social network: users create accounts on a Wafrn instance and can interact with both Fediverse and Bluesky users from a single identity [41].

Architecture. For AT Protocol support, Wafrn does not reimplement the protocol stack. Instead, it runs the official Bluesky PDS Docker image as a sidecar container and communicates with it through its stable APIs [31]. A Caddy reverse proxy routes requests between the main Wafrn domain and a separate PDS domain, with on-demand TLD certificate provisioning for per-user handle subdomains [31]. To receive events from the Bluesky network, a dedicated worker subscribes to Jetstream (a filtered, JSON-based projection of the AT Protocol firehose) and processes only events relevant to the instance's users [31].

ActivityPub is implemented directly in the backend, with routes for WebFinger, NodeInfo, actor endpoints, and inbox handling. Incoming activities are validated and enqueued for asynchronous processing via BullMQ, with the server returning an immediate HTTP 200 response to avoid blocking the sending server [31].

Identity and cross-posting. Each Wafrn user has a single database record that can carry both an ActivityPub actor URL and an AT Protocol DID [31]. Users who enable Bluesky integration in their profile settings receive an app password and a DID tied to the instance's PDS; their posts are then federated to both networks [41]. The posting pipeline is ordered deliberately: posts are first sent to the PDS, and only after the resulting

Bluesky URI is stored does ActivityPub federation begin. This allows the ActivityPub representation to include a reference to the Bluesky record, which downstream consumers can use for deduplication [31].

Comparison with Fedisky. Wafrn and Fedisky occupy different points in the design space. Wafrn is a full social networking application that happens to speak both protocols; Fedisky is a lightweight bridge that extends an existing AT Protocol PDS into the Fediverse without providing its own user interface or account system. A Wafrn deployment replaces both the social client and the server, while Fedisky is designed to be attached to an already-running PDS, requiring no changes to the user's existing Bluesky workflow.

The trade-off is scope versus simplicity. Wafrn offers a unified experience where both protocols are first-class citizens, including features like deduplication of cross-network identities. However, it requires users to adopt a new platform and its operators to maintain a full application stack. Fedisky requires no user migration: existing Bluesky users gain an ActivityPub presence automatically once the PDS operator deploys the sidecar, and their data remains in the AT Protocol repository they already use.

3 Design and Implementation

Fedisky is a sidecar service that a Bluesky PDS operator can deploy alongside their PDS to enable federation with the Fediverse. It acts as a bridge between the AT Protocol and ActivityPub, allowing users on Mastodon instances to discover, follow, and interact with users on the Bluesky PDS. Fedisky is designed to be a lightweight and modular service that can be easily deployed and maintained by PDS operators, without requiring any modifications to the PDS itself. In this chapter, we will provide an overview of the design and implementation of Fedisky, including its architecture, data model, key components, and operational considerations. Fedisky's source code is available on GitHub at `https://github.com/msonnb/fedisky`.

3.1 Overview

3.1.1 Technical Challenges

As outlined in the problem statement (Chapter 1), bridging AT Protocol and ActivityPub requires reconciling incompatibilities in identity, content representation, delivery mechanisms, and engagement semantics. Fedisky must bridge each of these gaps while remaining transparent to both sides—Bluesky users should not need to take any action, and Mastodon users should be able to interact with bridged accounts as if they were native ActivityPub actors. The following sections describe how Fedisky addresses each of these challenges in practice; Chapter 4 reflects on the trade-offs involved.

3.1.2 System Context

Figure 3.1 shows the wider system context of Fedisky and how it interacts with other systems and users. At the core of the system is the PDS host, which runs both the AT Protocol PDS and the Fedisky sidecar and is managed by the PDS operator. Fedisky uses AT Protocol XRPC APIs to read and write records from the PDS, and subscribe to its firehose endpoint to receive a real-time stream of all new and updated records.

Fedisky exposes ActivityPub endpoints, which are used by external Fediverse instances to deliver ActivityPub content to their users and to receive incoming activities from the Fediverse. These endpoints include a Webfinger endpoint for user discovery, an actor endpoint for representing Bluesky users as ActivityPub actors, and an inbox endpoint for receiving incoming activities from the Fediverse.

To fetch Bluesky records from users on other PDS instances, Fedisky fetches records from the Bluesky AppView. In addition, Fedisky periodically polls the AT Protocol

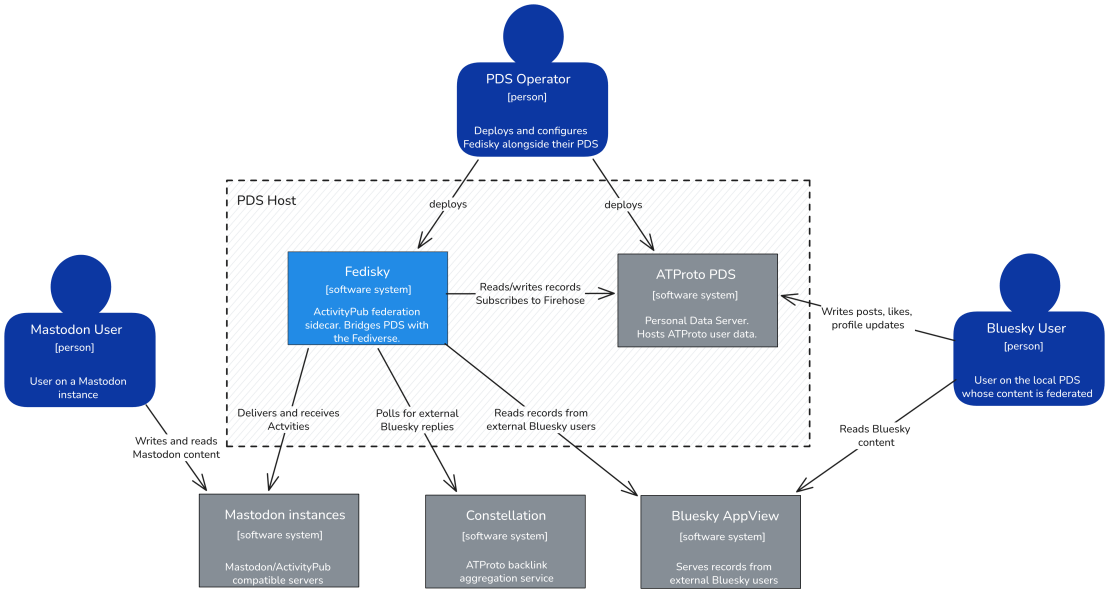

Figure 3.1: System Context Diagram showing how the Fedisky sidecar interacts with other systems and users.

Constellation API[1], an external service that aggregates and indexes backlinks across the entire AT Protocol, allowing Fedisky to discover interactions such as replies from users on other PDS instances, and federating them to the Fediverse.

3.1.3 Technology Stack

Fedisky is implemented in TypeScript and runs on Node.js. It stores its data in a SQLite database using Kysely[2] as a type-safe query builder. For federation functionality, Fedisky uses Fedify[3], a TypeScript library for building ActivityPub servers.

Fedify provides a high-level API for defining ActivityPub actors, registering dispatchers for handling incoming activities, and sending outgoing activities to other Fediverse instances. It also handles the underlying ActivityPub protocol primitives such as signing and verifying HTTP requests, providing type-safe objects for Activity Vocabulary such as `Create`, `Follow`, and `Note`. In addition, Fedify includes scalability and reliability features such as retry logic for failed deliveries, a message queue for processing incoming and outgoing activities, and a KV-store for caching and storing federation-related data such as public keys and remote actor information.

[1] `https://constellation.microcosm.blue/`
[2] `https://kysely.dev/`
[3] `https://fedify.dev/`

3.2 Architecture

Fedisky is structured around a set of loosely coupled subsystems, each responsible for a distinct concern. These subsystems are wired together at startup through a shared dependency injection container, `AppContext`, which holds references to all major services: the database, the PDS and AppView clients, the two bridge account managers, the Fedify federation instance, and the logger. The main service class, `APFederationService`, acts as the top-level orchestrator: it initializes the database, provisions the bridge accounts, starts the HTTP server, and conditionally launches the background processors.

3.2.1 Configuration

Fedisky uses a two-tier configuration system. The `readEnv()` function reads raw values from environment variables, and `envToConfig()` transforms them into a strongly typed `APFederationConfig` object, applying defaults where values are not provided. Only three variables are required, the PDS URL, PDS hostname, and PDS admin token. All other settings have sensible defaults derived from these.

The public URL is inferred from the hostname: if the hostname is `localhost`, the URL uses HTTP with the configured port; otherwise, it constructs an HTTPS URL, reflecting the assumption that production deployments will use TLS. Bridge account handles default to `mastodon.{hostname}` and `bluesky.{hostname}`, and their email addresses are generated as `noreply+{handle}@{hostname}`, since the PDS requires a unique email for each account. Background processors are enabled by default and their poll intervals are configurable. The Constellation processor defaults to 60 seconds, the DM notification processor to 5 minutes with a 10-minute batch delay. The database location defaults to an in-memory SQLite database, which is useful for testing but must be overridden with a file path for production use.

An `allowPrivateAddress` flag, disabled by default, permits Fedify to fetch resources from private IP ranges. This is necessary for the end-to-end test environment, where all services run on local addresses, but must remain disabled in production to prevent Server-Side Request Forgery (SSRF) attacks. A complete list of all supported environment variables and their defaults is provided in Section A.1.

3.2.2 PDS and AppView Clients

The `PDSClient` and `AppViewClient` modules provide thin wrappers around the AT Protocol XRPC APIs. The PDS client is used for all write operations and local record lookups as well as identity resolution and blob retrieval. The AppView client is used exclusively for read operations against the Bluesky AppView, such as fetching posts from users on external PDS instances that are not accessible through the local PDS. Separating the two clients reflects the architectural distinction in AT Protocol between the PDS as the authoritative data store and the AppView as a read-optimized aggregation layer.

3.2.3 Firehose Processor

The `FirehoseProcessor` subscribes to the PDS's `com.atproto.sync.subscribeRepos` WebSocket endpoint and drives the outbound federation pipeline. Each incoming frame is decoded from its CBOR binary encoding and parsed into a structured commit event containing the repository DID, a sequence number, and a list of operations. The processor filters operations to only those affecting collections for which a converter is registered, and skips events originating from the bridge accounts to prevent bridging loops. For create operations, it fetches the complete record from the PDS, converts it into an ActivityPub activity using the converter registry, and dispatches it to followers via Fedify. For delete operations, it constructs the corresponding `Delete` or `Undo` activity and dispatches it similarly. The processor also adds newly created local posts to the `ap_monitored_post` table for later processing by the Constellation processor. On connection loss, the processor waits five seconds before automatically reconnecting.

3.2.4 Constellation Processor

The `ConstellationProcessor` runs as a periodic background job, polling at a configurable interval (default 60 seconds). On each run, it fetches a batch of up to 50 entries from the `ap_monitored_post` table, which tracks local posts that have been bridged to the Fediverse and may have received replies from users on other PDS instances. For each post, it queries the Constellation API for backlinks of type `app.bsky.feed.post reply.parent.uri`, which identifies posts on external PDS instances that reply to the monitored post. Each discovered reply is checked against the `ap_external_reply` deduplication table. New replies are fetched from the AppView, converted into `Create(Note)` activities attributed to the Bluesky bridge account, and delivered to the original post author's followers via Fedify. The `ap_external_reply` record is then created and the monitored post's `lastChecked` timestamp is updated.

3.2.5 DM Notification Processor

The `DMNotificationProcessor` polls the database at a configurable interval (default 5 minutes) for likes and reposts that have not yet triggered a notification and whose timestamp is older than a configurable batch delay (default 10 minutes), ensuring that a burst of engagements on a single post results in a single summary message rather than individual notifications. Engagements are grouped by post author and then by post, and a summary direct message is sent to each author via the Bluesky Chat API using the Mastodon bridge account as the sender. The message includes the display names of the engaging Fediverse actors, resolved by fetching their ActivityPub profiles, and a truncated excerpt of the post content.

3.2.6 Database

All persistent state is stored in a SQLite database through Kysely. A second SQLite database is used exclusively by Fedify for its internal KV store and message queue. Schema

changes are managed through numbered migration files, each exporting `up()` and `down()` functions, which are applied automatically on service startup.

Identity & Cryptography

- `ap_key_pair` – Stores the cryptographic key pairs used for HTTP signature signing. Each local PDS user gets two key pairs generated on first access: one RSA for compatibility with older ActivityPub implementations, and one Ed25519 for modern servers. The keys are stored as PEM-encoded strings.
- `ap_bridge_account` and `ap_bluesky_bridge_account` – Singleton tables that store the credentials for the two bridge accounts (see Section 3.2.7).

Social Graph

- `ap_follow` – Records which ActivityPub actors follow which local PDS users. This is the core table for activity delivery, as it determines which users should receive which activities based on their follow relationships. It stores the follower's inbox URL and shared inbox URL for efficient delivery.

Content Mapping

- `ap_post_mapping` – Maps AT Protocol post URIs to their original ActivityPub Note IDs and author information. This table is essential for correct reply threading.
- `ap_external_reply` – Stores external Bluesky replies from other PDS instances that have been federated via the Constellation processor. Primarily for deduplication as the Constellation API is polled repeatedly.
- `ap_monitored_post` – The work queue for the Constellation processor, which tracks which posts need to be checked for new external replies.

Engagement Tracking

- `ap_like` and `ap_repost` – Stores incoming ActivityPub `Like` and `Announce` activities targeting local posts. The tables' main purpose is to track engagements for display (e.g. showing like counts) and batching engagement for DM notifications.

3.2.7 Bridge Account System

Fedisky uses two special "bridge accounts" to facilitate bridging between the AT Protocol and ActivityPub. The first is the Mastodon bridge account, which is hidden from users in ActivityPub and is used to post incoming Fediverse replies as AT Protocol posts on the PDS, so that they appear in the Bluesky user's thread and can be replied to and interacted with like normal posts. Its handle and display name can be configured by the operator, with the default being `mastodon.{hostname}` and "Mastodon Bridge" respectively.

The second one is the Bluesky bridge account, which is used to federate replies from Bluesky users on other PDS instances to the Fediverse. This allows users on Mastodon to see and interact with replies from users on other PDS instances, which would otherwise be invisible to the Fediverse. The Bluesky bridge account's default handle is `bluesky.{hostname}` and display name is "Bluesky Bridge", again configurable by the operator via environment variables.

Account Lifecycle. The two bridge accounts are managed by the `MastodonBridgeAccountManager` and `BlueskyBridgeAccountManager`, both subclassing an abstract `BridgeAccountManager`. On startup, each manager checks the database for stored credentials. If found, it attempts to refresh the session, falling back to a password login if the refresh token is expired. If no account record exists, it creates a new PDS account using the configured handle and a random password. Once initialized, each manager exposes the account's DID and a valid access token for use in XRPC calls. The Mastodon bridge account is hidden from ActivityPub discovery, while the Bluesky bridge account is exposed as a regular actor.

Attribution Model. Since both bridge accounts post content on behalf of users and do not carry any user identity in their handle or display name, we need to ensure proper attribution of content to the original authors. For incoming Fediverse replies posted by the Mastodon bridge account, we include the original author's handle in the first line of the post content, e.g. "@bob@mastodon.social replied:", followed by the actual reply content. Similarly, federated Bluesky replies from third-party PDS instances sent by the Bluesky bridge account include an attribution line with the original author's Bluesky handle, e.g. "alice.bsky.social replied:", followed by the reply content. In both cases, the handle is a clickable link to the original profile, allowing users to easily find and follow the original author if they wish. Additionally, since Mastodon content is formatted in HTML, we need to ensure all content is properly escaped to prevent Cross-Site Scripting (XSS) vulnerabilities.

3.3 Core Flows

3.3.1 Actor and Identity Discovery

When a Mastodon user tries to follow a federated Bluesky user on the PDS, the Mastodon instance needs to discover the corresponding ActivityPub actor for that user in order to send the follow request. This flow is illustrated in Figure 3.2. The Mastodon instance first queries the WebFinger endpoint with the user's handle (e.g. `@alice@fedisky.social`) to discover the corresponding ActivityPub actor URL. Fedisky first constructs the AT Protocol handle by prepending the localpart (in this case, `alice`) to the PDS's hostname (e.g. `fedisky.social`), resulting in `alice.fedisky.social`. Note that the ActivityPub handle domain and PDS domain do not have to match, but in this example we use the same domain for simplicity. Fedisky then resolves this handle using the PDS's

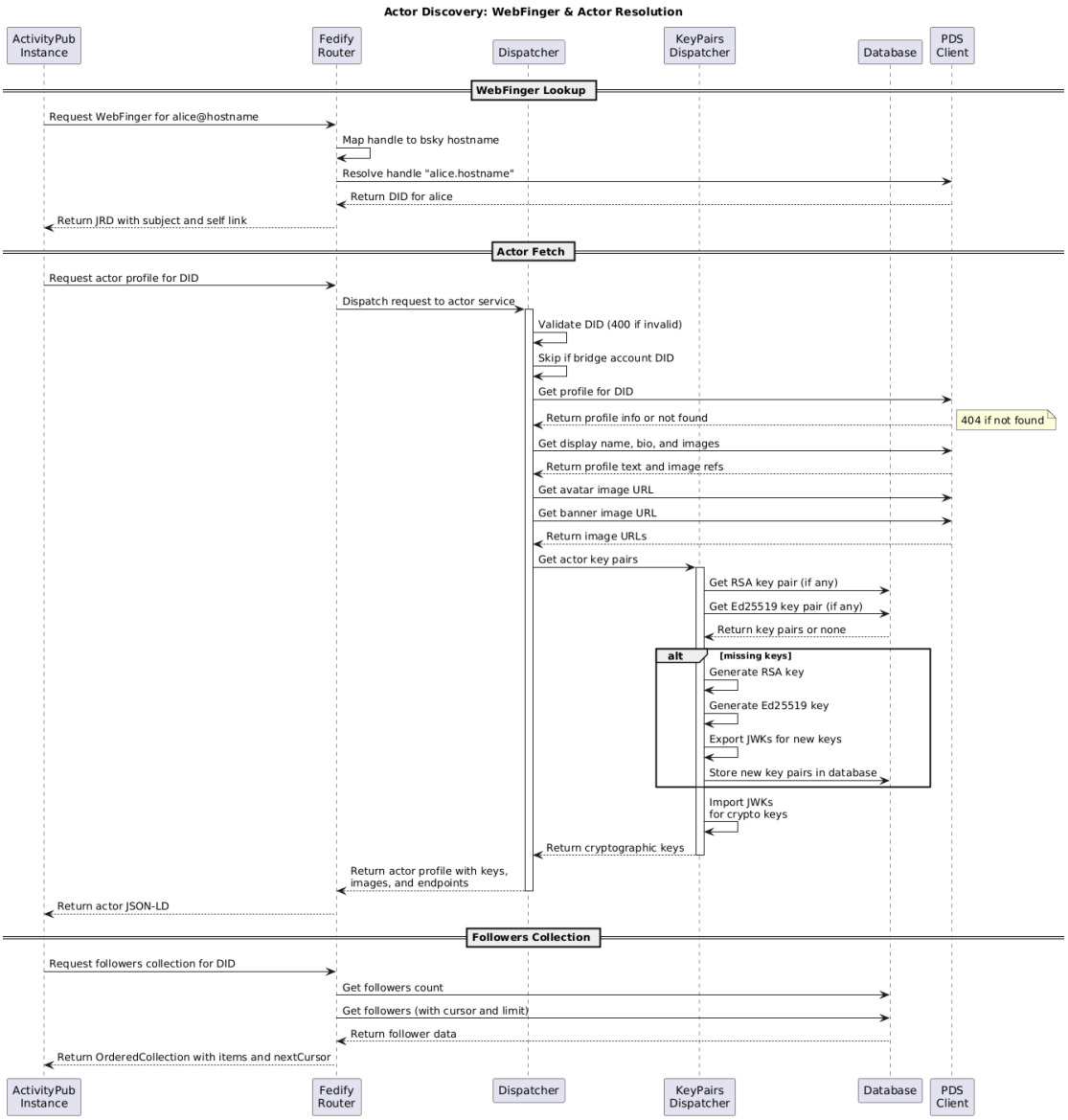

Figure 3.2: Sequence diagram showing the actor and identity discovery flow when a Mastodon user tries to follow a Bluesky user on the PDS.

`com.atproto.identity.resolveHandle` API, which returns the corresponding DID if a matching user is found. If a user is found, Fedisky constructs the ActivityPub actor URL using the user's DID, resulting in `https://fedisky.social/users/{did}`. This URL is returned to the Mastodon instance in the WebFinger response, as shown in Listing 3.1. In addition to the actor URL, the response also includes references to the user's profile page and avatar.

```
{
  "subject": "acct:alice@fedisky.social",
  "aliases": ["https://fedisky.social/users/did:plc:
    n3jcidccul6u3lif5q4rh42x"],
  "links": [
    {
      "rel": "self",
      "href": "https://fedisky.social/users/did:plc:
    n3jcidccul6u3lif5q4rh42x",
      "type": "application/activity+json"
    },
    {
      "rel": "http://webfinger.net/rel/profile-page",
      "href": "https://bsky.app/profile/alice.fedisky.social"
    },
    {
      "rel": "http://webfinger.net/rel/avatar",
      "href": "https://fedisky.social/xrpc/com.atproto.sync.getBlob?did=
    did%3Aplc%3An3jcidccul6u3lif5q4rh42x&cid=
    bafkreiahfo2qkotgr475q2kx4psbffojwup6fkchnw43y2i44uhanmy2em",
      "type": "image/jpeg"
    }
  ]
}
```

Listing 3.1: WebFinger response for `@alice@fedisky.social`.

After receiving the WebFinger response, the Mastodon instance can then query the actor endpoint to fetch the user's actor document, which includes the user's profile information, public keys, and inbox URLs. The object conforms to the Activity Vocabulary `Person` type as defined in [36]. To construct the actor document, Fedisky fetches the user's profile record from the PDS using the `com.atproto.repo.getRecord` API. From the profile record, Fedisky extracts the user's display name, description, as well as avatar and banner images. Finally, Fedisky retrieves the user's public keys from the database. If no keys exist yet, Fedisky generates new RSA and Ed25519 key pairs and stores them in the database.

In an effort to link accounts referring to the same identity, Barrett [4] proposes the use of account links. In this approach, instead of relying on a separate "meta account" that links all the user's accounts together, accounts reference each other. Platforms can then use these references to show highlighted links to these other accounts on different platforms. Fedisky follows this approach and includes the user's AT Protocol URI in the `alsoKnownAs` field of the actor document, allowing Fediverse instances to link the ActivityPub actor back to the original Bluesky user and profile.

3.3.2 Outbound: AT Protocol Post to ActivityPub Note

When a Bluesky user creates or deletes a post on the PDS, Fedisky propagates these changes to the Fediverse so that Mastodon users following that user receive updates. This flow is illustrated in the sequence diagram in Figure 3.3.

Fedisky receives real-time updates from the AT Protocol firehose, a WebSocket stream that delivers commits as CBOR-encoded frames. Each frame contains repository operations (create, update, or delete) along with metadata such as the repository DID and sequence number. The firehose processor decodes the CBOR frame and extracts the repository DID, operations, and sequence information. It then filters operations to only process those that represent record creation or deletions for collections that Fedisky can convert to ActivityPub activities.

For create operations, the processor first fetches the complete record from the PDS using the `com.atproto.repo.getRecord` API, providing the repository DID, collection identifier (e.g. `app.bsky.feed.post`), and record key. Once the record is retrieved, the processor queries the record converter registry to obtain the appropriate converter for the collection type. The registry returns a converter instance that implements the conversion logic for that specific AT Protocol record type.

The converter then transforms the AT Protocol record into an ActivityPub `Create` activity containing a `Note` object. During conversion, if the post is a reply, the converter queries the database to look up the parent's post ActivityPub mapping, which maps the AT Protocol URI to the corresponding ActivityPub object ID and actor inbox. This mapping is necessary to construct the `inReplyTo` property of the ActivityPub Note. If the post contains embedded images or media, the converter builds the blob URLs using the PDS's `com.atproto.sync.getBlob` API, and includes them as attachments in the Note. Additionally, the converter transforms plain text into HTML and AT Protocol self-labels into ActivityPub content warnings if necessary. A detailed explanation of the conversion logic can be found in Section 3.5.

After conversion completes, Fedisky stores the post mapping in the database, recording the AT Protocol URI, author DID, and creation timestamp. This mapping enables Fedisky to track which posts have been federated and to handle subsequent operations like deletions. If the post is a reply to a bridge account post (i.e. a post originally created by a Fediverse user and relayed back to Bluesky), Fedisky performs an additional step: It looks up the bridge post mapping to retrieve the bridge actor's ID and inbox URL, then sends the `Create` activity directly to that actor's inbox. This ensures that the original Fediverse author receives notifications about replies to their content.

Finally, Fedisky delivers the `Create` activity to all the author's federated followers. The Fedify router queries the database to retrieve the list of followers and for each follower, it determines the appropriate inbox URL (either the follower's personal inbox or their instance's shared inbox) and sends a request containing the activity to that inbox endpoint. The activity is signed using the author's private key to authenticate the request according to ActivityPub's HTTP signature specification.

For delete operations, the flow is simpler. The processor constructs either a `Delete` activity (for post deletions) or an `Undo` activity (for undoing a like or repost) targeting the

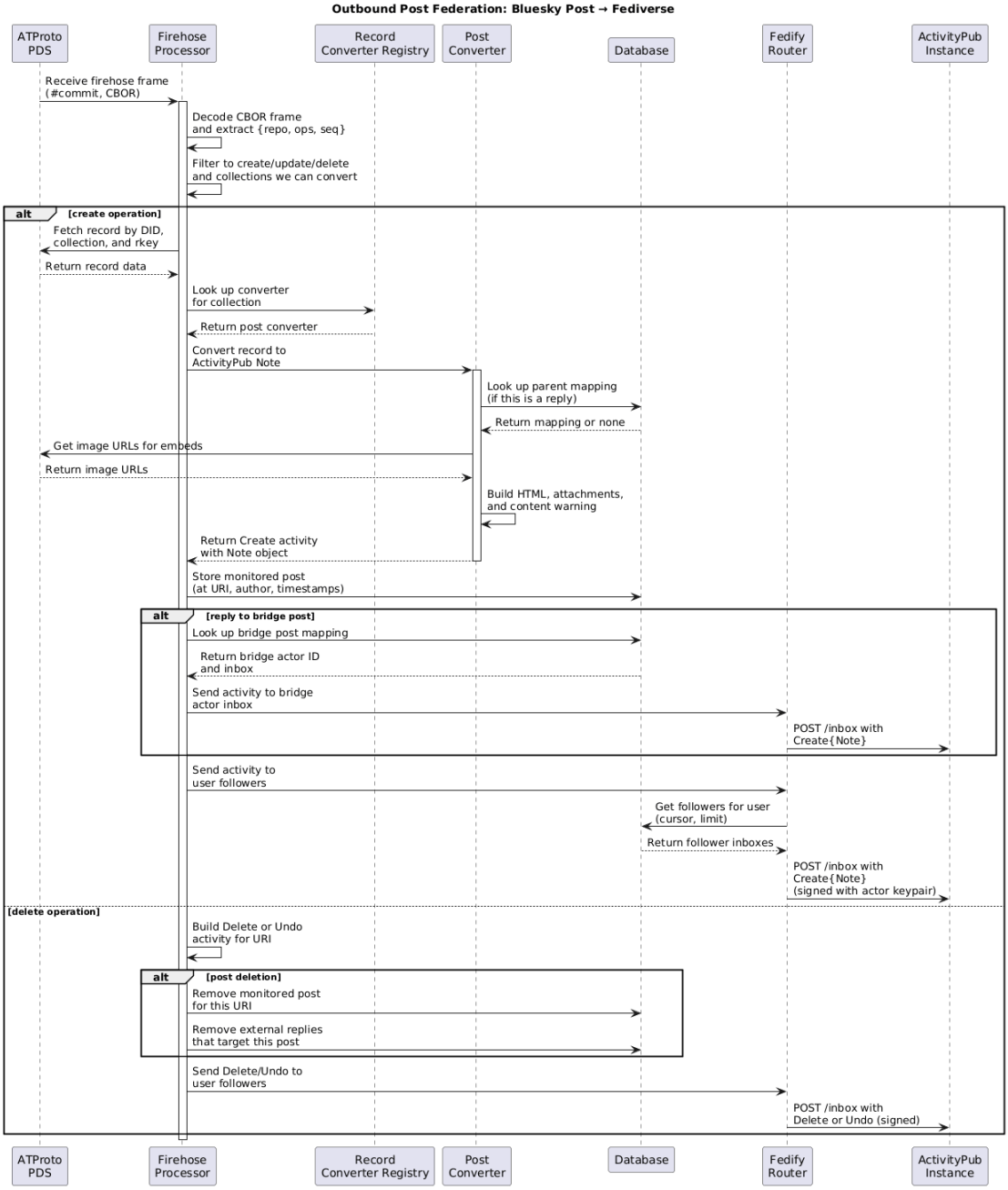

Figure 3.3: Sequence diagram showing the outbound AT Protocol post to ActivityPub note conversion flow.

ActivityPub object corresponding to the deleted AT Protocol record. If the deletion is for a post, Fedisky also cleans up any associated database entries related to that post. The `Delete` or `Undo` activity is then delivered to all followers using the same inbox delivery mechanism as create operations, ensuring that federated instances are notified of the deletion.

3.3.3 Inbound: ActivityPub Activity to AT Protocol Record

Figure 3.4 shows the inbound flow for incoming ActivityPub activities. All incoming activities from the Fediverse are delivered to Fedisky's inbox endpoints by the remote ActivityPub instance. Fedify first verifies the HTTP signature on each request and fetches the sending actor from the network before dispatching the activity to the appropriate handler. The handling logic then branches based on the activity type as follows:

Follow. When Fedisky receives a `Follow` activity, it validates the activity's required fields (identifier, actor, and object), then parses the object URI to extract the target user's DID. The follow relationship is persisted to the database, recording the follower's actor URI and their personal and shared inbox URLs for later use during activity delivery. Fedisky then sends an `Accept(Follow)` activity back to the actor's inbox, completing the handshake and activating the follow relationship on the remote instance.

Undo(Follow), Undo(Like), Undo(Announce). These three undo variants are handled symmetrically. Fedisky locates the original record in the database by actor URI or activity ID, and removes it. No further action is required on the PDS.

Delete(Actor). When an actor deletion is received, Fedisky performs a full cleanup of all data associated with that actor. It removes all follow, like, and repost records from the database. It then queries the database for all post mappings belonging to that actor, iterating over each bridged post and issuing a `com.atproto.repo.deleteRecord` call via the bridge account client to remove the corresponding AT Protocol record from the PDS. Finally, all remaining post mappings for the actor are removed from the database.

Delete(Note). For individual note deletions, Fedisky looks up the post mapping by the ActivityPub note ID to retrieve the corresponding AT Protocol collection and record key. If a mapping is found, Fedisky issues a `com.atproto.repo.deleteRecord` call to the PDS to remove the record, then removes the post mapping entry from the database.

Like and Announce. When a `Like` or `Announce` activity is received, Fedisky parses the object URI to determine the target post's AT Protocol URI and the author's DID. It then queries the PDS to verify that an account with that DID is local to the PDS, discarding the activity if not. If the post is confirmed to be local, the engagement is stored in the database in the respective `ap_like` or `ap_repost` table, recording the activity ID, post URI, and actor URI.

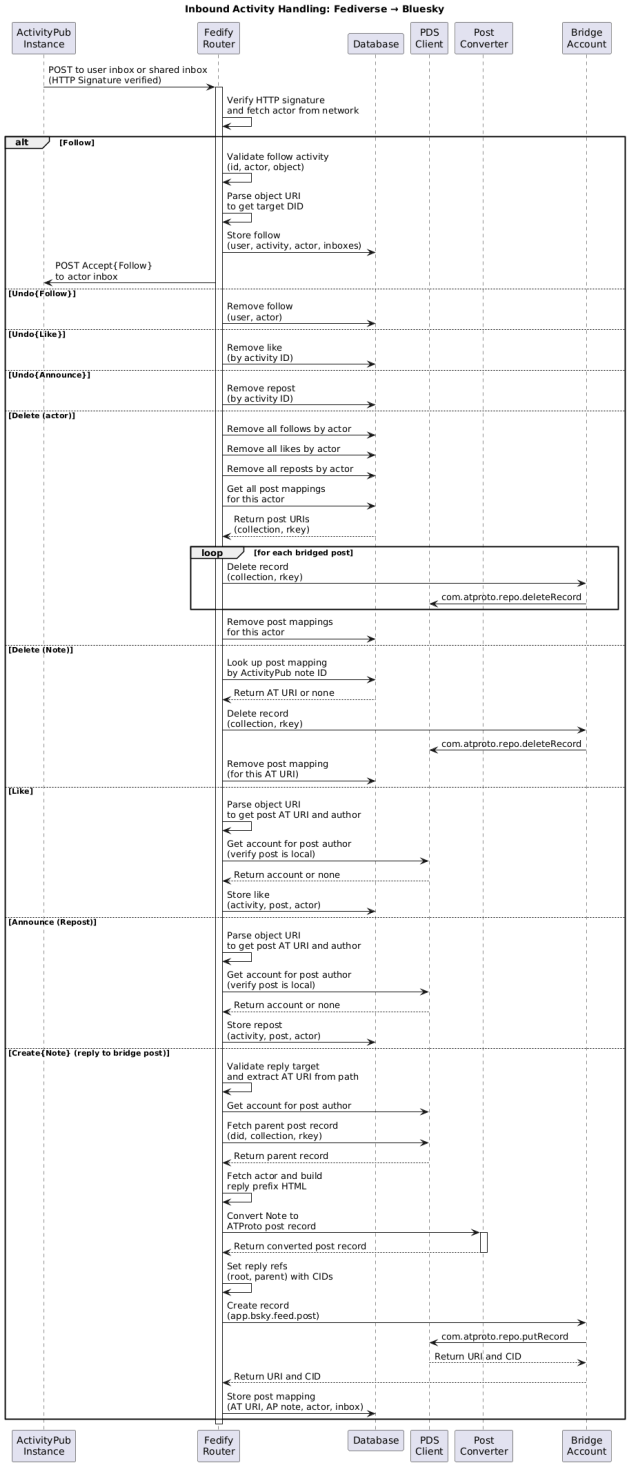

Figure 3.4: Sequence diagram showing the inbound ActivityPub activity to AT Protocol record conversion flow.

Create(Note). The most involved inbound case handles incoming Fediverse replies to posts that were originally bridged onto the PDS by the Mastodon bridge account. Fedisky first validates the reply target and extracts the parent post's AT Protocol URI from the path. It then fetches the account for the post's author and retrieves the full parent post record from the PDS using the `com.atproto.repo.getRecord` API, obtaining the record's Content Identifier (CID) required for correct reply threading. Next, Fedisky fetches the remote actor's profile and constructs an attribution prefix in HTML (e.g. "@bob@mastodon.social replied:") to be prepended to the post content. The `Note` is then passed to the post converter, which transforms it into an AT Protocol `app.bsky.feed.post` record. After conversion, Fedisky sets the reply references (`root` and `parent`), resolving the CIDs of both the root and parent records so that the reply is correctly threaded in the Bluesky client. The record is then written to the PDS via the `com.atproto.repo.putRecord` API using the bridge account client, which returns the new record's URI and CID. Finally, Fedisky stores a post mapping in the database, recording the AT Protocol URI, the original ActivityPub note ID, the actor URI, and the actor's inbox URL, enabling future operations such as deletions to locate the bridged post.

3.4 ActivityPub Interface

Using Fedify together with its Express HTTP integration[4], Fedisky exposes the following ActivityPub endpoints. Fedify handles the low-level protocol concerns: verifying and signing HTTP requests, and managing the message queue and KV store backed by a dedicated SQLite database. The application layer registers dispatchers that Fedify calls for each activity type or collection lookup, which in turn delegate to the appropriate database queries or converter logic.

3.4.1 Endpoints

- `GET /.well-known/webfinger`
  The WebFinger endpoint for user discovery. When a Mastodon instance encounters a handle such as `@alice@fedisky.social`, it will query this endpoint to discover the corresponding ActivityPub actor URL. Fedisky resolves the handle using the PDS's `com.atproto.identity.resolveHandle` API, and if a matching PDS user is found, constructs an ActivityPub actor URL using the user's DID as the unique identifier, i.e. `https://fedisky.social/users/{did}`.

- `GET /users/{did}`
  The actor endpoint. Returns an ActivityPub `Person` object representing the Bluesky user with the given DID. Includes the user's inbox, outbox, followers, and following URIs, as well as their public keys (both RSA and Ed25519), and profile information such as display name and avatar. It also includes a `alsoKnownAs` field with the

[4] `https://fedify.dev/manual/integration#express`

user's AT Protocol URI, in order to link the ActivityPub actor back to the original PDS user and identity. The Mastodon bridge account does not have an ActivityPub actor representation.

- `POST /users/{did}/inbox`
  The inbox endpoint for receiving incoming activities from the Fediverse. This is where we receive activities such as `Follow` requests. When an activity is received, we verify the HTTP signature to ensure it is from a trusted source, and then dispatch it to the appropriate handler based on the activity type.

- `POST /inbox`
  A server-wide shared inbox endpoint that some Fediverse instances support for more efficient delivery. Fedisky also supports this endpoint for incoming activities, and dispatches them in the same way as the user-specific inbox.

- `GET /users/{did}/outbox`
  Paginated collection of posts, likes, and reposts, aggregated from the PDS using its `com.atproto.repo.listRecords` API. This allows Mastodon instances to fetch the user's content and engagements for display on their profile and timelines.

- `GET /users/{did}/followers` and `GET /users/{did}/following`
  Paginated collections of followers and following, based on the `ap_follow` table. This allows Mastodon instances to display the user's followers and following lists, and to determine which users they should receive activities from.

- `GET /posts/{uri}`
  Endpoint for fetching a specific post by its AT Protocol URI, used by Mastodon instances to fetch the content and metadata of a post when displaying it or when a user clicks on a link to the post. Fedisky resolves the AT Protocol URI using the PDS's `com.atproto.repo.getRecord` API, and returns an ActivityPub Note object with the post content, author information, and any media attachments.

- `GET /nodeinfo/2.1`
  The NodeInfo[5] endpoint providing metadata about the Fedisky instance, such as software name and version, and supported features. This is used by Mastodon instances to determine compatibility and capabilities of the Fedisky bridge.

3.4.2 Outbox

The outbox dispatcher aggregates records from all three bridged AT Protocol collections (posts, likes, and reposts) into a single, chronologically ordered ActivityPub outbox collection. For a given user, it queries the PDS's `com.atproto.repo.listRecords` API for each collection registered in the converter registry, then merges the results into a unified list. To produce a consistent chronological ordering across collections, the records

[5] `https://nodeinfo.diaspora.software/`

are sorted by their record key, which in AT Protocol is a Timestamp Identifier (TID) that sorts lexicographically by creation time. The outbox is paginated using cursor-based pagination: each page contains up to 50 items, and the cursor for the next page is derived from the last record's key. An extra record is fetched beyond the page limit to determine whether additional pages exist.

Each record in the page is passed to its corresponding converter, which transforms it into an ActivityPub activity. Records whose conversion fails are filtered from the response, and errors logged via the wide events system. A separate object dispatcher handles direct lookups of individual posts by their AT Protocol URI. When a Mastodon instance fetches a post URL (e.g. `https://fedisky.social/posts/{uri}`), the dispatcher parses the AT Protocol URI from the path, fetches the record from the PDS, and returns the converted ActivityPub `Note` object.

3.5 Conversion Layer

The conversion layer provides bidirectional translation between AT Protocol records and ActivityPub objects. A `RecordConverterRegistry` maps AT Protocol collection identifiers (e.g. `app.bsky.feed.post`) to converter instances implementing a common `RecordConverter` interface. Each converter exposes two methods: `toActivityPub`, which transforms an AT Protocol record into an ActivityPub activity and/or object, and `toRecord`, which performs the inverse transformation. The registry is populated at startup with converters for the three bridged collection types: posts, likes, and reposts.

3.5.1 Post Converter

The post converter handles the `app.bsky.feed.post` collection and is the most involved of the three converters, as it must handle bidirectional conversion including rich text, media, reply threading, mentions, and content warnings.

Outbound (AT Protocol to ActivityPub). The `toActivityPub` method receives an AT Protocol post record and constructs a `Create(Note)` activity. The note's ID is derived from the AT Protocol URI, and its `to` field is set to the ActivityPub public collection to make the post visible to all Fediverse users. The author's followers URI is included in the `cc` field to ensure delivery.

If the post is a reply, the converter queries the database for the parent post's ActivityPub mapping. If a mapping exists (i.e. the parent was originally an ActivityPub note bridged onto the PDS), the original ActivityPub note ID is used as the `inReplyTo` value, preserving the correct reply threading on the Fediverse side. Otherwise, the converter constructs an ActivityPub object URI from the AT Protocol parent URI.

The post's plain text is converted to HTML by splitting on double newlines to produce `<p>` elements, with all content escaped using the `escape-html` NPM package[6] to prevent XSS vulnerabilities.

[6] `https://www.npmjs.com/package/escape-html`

Inbound (ActivityPub to AT Protocol). The `toRecord` method receives an ActivityPub note object and transforms it into an `app.bsky.feed.post` record. The HTML content is parsed into plain text using the `html-to-text` NPM package[7] (see Section 3.5.3 below), and the resulting text is truncated to 3000 bytes if necessary to respect Bluesky's post size limits [21]. If the note includes attachments, they are downloaded and uploaded to the PDS as blobs (see Section 3.5.4). If the note specifies a `replyTarget`, the converter parses the AT Protocol URI from the ActivityPub URL to construct the reply reference. A record key is generated using AT Protocol's TID scheme, and the record's CID is computed using Lexicon CBOR hashing.

3.5.2 Like and Repost Converters

The like and repost converters handle the `app.bsky.feed.like` and `app.bsky.feed.repost` collections respectively. Both converters are outbound-only: they implement `toActivityPub` but return null from `toRecord`, since incoming likes and reposts from the Fediverse are handled directly by the inbox handlers rather than through the conversion layer.

The like converter transforms an AT Protocol like record into an ActivityPub `Like` activity, while the repost converter produces an `Announce` activity. Both converters first check whether the subject post belongs to a local PDS user using the `isLocalUser` utility, which queries the PDS for the account. If the subject post is from an external PDS, the activity is discarded to avoid generating engagement notifications for posts that are not managed by this Fedisky instance.

3.5.3 HTML ↔ Rich Text

AT Protocol and ActivityPub use fundamentally different text representations. AT Protocol stores post content as plain text with separate annotations called *facets*, which specify byte ranges and their semantic type (link, mention, or tag). ActivityPub, and more specifically Mastodon, stores content as HTML with inline anchor elements for links and mentions. When converting AT Protocol posts to ActivityPub notes, the converter splits the plain text on double newlines and wraps each paragraph in `<p>` tags. All text content is HTML-escaped to prevent injection of arbitrary markup.

The inverse conversion uses the `html-to-text` library with a custom `anchorCollector` formatter. This formatter walks the HTML DOM tree and, for each anchor element, extracts both the visible text content and the `href` URL. It also inspects the element's CSS classes: Mastodon marks mention links with the `mention` class, which allows the converter to distinguish mentions from regular hyperlinks. Elements with the `invisible` class (used by Mastodon to hide portions of long URLs for display purposes) are skipped during text extraction to produce clean output. The extracted links are collected separately and used in a subsequent step to reconstruct AT Protocol facets.

After HTML parsing, the converter reconstructs AT Protocol facets from the collected links. For regular hyperlinks, it locates the link's visible text within the plain text

[7] `https://www.npmjs.com/package/html-to-text`

output using positional search, then computes the UTF-8 byte offsets required by AT Protocol's facet index format. For mention links, the converter extracts the DID from the ActivityPub actor URL (which follows the pattern `/users/{did}`), verifies that the mentioned user is local to the PDS, and constructs a mention facet with the resolved DID.

If the ActivityPub note specifies a language via the `LanguageString` type, the converter extracts it and includes it in the AT Protocol record's `langs` array, preserving language information across the bridge.

3.5.4 Media Handling

Outbound. When an AT Protocol post contains embedded images or video, the converter constructs ActivityPub `Document` attachments by building blob URLs using the PDS's `com.atproto.sync.getBlob` endpoint. Each attachment includes the blob's MIME type and alt text. The converter supports both the `app.bsky.embed.images` type and the `app.bsky.embed.video` type.

Inbound. For incoming ActivityPub notes with attachments, the converter downloads each remote media file and uploads it to the PDS as a blob. The blob handler enforces several security and resource constraints: it validates that URLs use HTTP or HTTPS and rejects private IP addresses (RFC 1918 ranges, localhost, and link-local addresses) to prevent SSRF attacks. Downloads are subject to a 30-second timeout and a 10 MB size limit, enforced both via the `Content-Length` header and by checking the actual downloaded data size. The downloaded blobs are then classified by MIME type: image blobs are mapped to `app.bsky.embed.images` (limited to four images, matching Bluesky's constraint), while video blobs are mapped to `app.bsky.embed.video` (limited to a single video) [21, 19, 20].

3.5.5 Content Warning and Label Mapping

AT Protocol and ActivityPub represent content sensitivity differently. AT Protocol uses *self-labels*, machine-readable string identifiers such as `porn`, `sexual`, `nudity`, and `graphic-media` that moderation systems use to filter or blur content [16]. ActivityPub uses a boolean `as:sensitive` flag and a free-text `summary` field that Mastodon displays as a content warning [30].

When an AT Protocol post carries self-labels, the converter maps each recognized label to a human-readable description (e.g. `porn` becomes "Adult Content (Porn)"). The descriptions are joined into a comma-separated string and set as the note's `summary`, with the `as:sensitive` flag set to true. This allows Mastodon to display an appropriate content warning and give users the option to view the content if they choose.

In the reverse direction, the converter scans the content warning text for keywords using regular expressions (e.g. matching "nsfw", "adult content", or "explicit" to the `sexual` label). If the `as:sensitive` flag is set, but no keywords match, the converter defaults to the `sexual` label as a general-purpose fallback. The matched labels are assembled into an

AT Protocol structure with the `com.atproto.label.defs#selfLabel` type and attached to the post record.

3.6 Observability & Operations

3.6.1 Wide Events Logging

Fedisky implements the *wide events* pattern, where each request or background operation emits a single, context-rich structured log event at completion rather than scattering multiple log statements throughout the code path. This pattern enables powerful debugging and analytics because all relevant context for a given operation is contained in one log line, making it straightforward to filter, search, and correlate events [22, 29, Ch. 5].

The implementation centers on the `WideEvent` class, a builder that accumulates key-value pairs over the lifetime of an operation. It supports dot-notation for nested fields (e.g. `actor.did`, `activity.type`), dedicated methods for attaching user context (`setUser`), error information (`setError`), and outcome classification (`setOutcome`, which accepts `success`, `error`, or `ignored`). Each event automatically records a timestamp at creation and computes the operation's duration at emission time. Environment metadata such as the service name, package version, Git commit hash, and Node.js version is captured once at startup and included in every event, ensuring that log lines can be correlated with specific deployments.

For HTTP requests, an Express middleware creates a `WideEvent` of type `http_request` at the start of each request, populating it with the HTTP method, path, user agent, and remote address. A request ID is extracted from standard tracing headers (`X-Request-Id`, `X-Correlation-Id`, `X-Trace-Id`, `Traceparent`), or generated as a UUID if none is present. The event is then made available to all downstream handlers and middleware through Node.js's `AsyncLocalStorage`, which propagates the event through the entire asynchronous call chain without requiring explicit parameter passing. Any handler in the request chain can retrieve the current event via `getWideEvent()` and enrich it with domain-specific fields. When the response finishes, the middleware automatically sets the HTTP status code and outcome, and emits the event. For background operations such as firehose message processing, events are created directly using `createWideEvent()`, enriched throughout the operation, and explicitly emitted upon completion.

Fedisky uses LogTape[8] as its logging framework, configured with two sinks: a console sink for local development, and an OpenTelemetry sink that exports structured logs to an observability backend. The OpenTelemetry integration is activated by importing the OpenTelemetry auto-instrumentation script at startup, which enables automatic instrumentation of HTTP requests, database calls, and other supported libraries like Fedify without code changes. This allows operators to connect Fedisky to any OpenTelemetry-compatible backend for distributed tracing and log aggregation.

[8] `https://logtape.org/`

3.6.2 HTTP Security

Fedisky applies several HTTP-level security measures. The `helmet`[9] middleware sets security-related HTTP headers (such as `X-Content-Type-Options`, `Strict-Transport-Security`, and `Content-Security-Policy`) to mitigate common web vulnerabilities. Request body size is limited to 256 KB to prevent denial-of-service through oversized payloads. Two rate limiters are applied: a general limiter allowing 1000 requests per 15-minute window, and a stricter inbox-specific limiter allowing 100 requests per minute per IP address, since the inbox endpoints are the primary target for unsolicited traffic from the Fediverse. Rate limit headers allow clients to inspect their remaining quota.

3.6.3 Testing

Unit Tests. The unit test suite uses Vitest[10] as the test runner and covers the core logic modules of the system: the conversion layer (post, like, and repost converters), the wide events logging system, the database layer, the firehose processor, the federation inbox handler, and the DM notification system. Tests are co-located with their source code in `tests/` subdirectories within each module. The conversion tests verify bidirectional translation correctness—for example, that an AT Protocol post with facets, embeds, and content labels is correctly converted to an ActivityPub note and back.

End-to-End Tests. The end-to-end test suite validates complete federation flows across service boundaries. The test environment is orchestrated by a shell script that provisions the following infrastructure:

- A real AT Protocol PDS instance running in Docker, configured with a test hostname (`bsky.test`).
- A Caddy reverse proxy providing TLS termination with locally-trusted certificates, since ActivityPub requires HTTPS for HTTP signatures and actor resolution.
- A mock ActivityPub server that simulates a Mastodon instance (`mastodon.test`), capable of sending and receiving ActivityPub activities and exposing an API for test assertions.
- A mock Constellation server that simulates the external reply discovery service.
- A native Fedisky instance connected to the PDS and configured with an in-memory database.

The test suite covers four categories of federation flows: actor discovery (WebFinger resolution and ActivityPub actor retrieval), follow federation (follow request delivery and acceptance), post federation (outbound post delivery to followers via the firehose pipeline), and Constellation-based external reply federation. Each test creates fresh PDS

[9] `https://www.npmjs.com/package/helmet`
[10] `https://vitest.dev`

accounts, establishes follows between mock ActivityPub users and PDS users, performs actions (such as creating posts), and then asserts that the expected ActivityPub activities were delivered to the mock server. Tests use polling with timeouts to account for the asynchronous nature of the system, where activities flow through the firehose, conversion layer, and HTTP delivery pipeline before arriving at the remote server.

3.6.4 Deployment

Fedisky is packaged as a multi-architecture Docker image (AMD64 and ARM64) and published to the GitHub Container Registry. The CI/CD pipeline, implemented as a GitHub Actions workflow, builds and pushes the image on pushes to the `release` branch or on version tags. Images are tagged with the semantic version, the `major.minor` version, the full Git SHA, and `latest`.

The production container image uses a multi-stage build: a build stage compiles TypeScript and installs dependencies, and a production stage copies only the compiled output and production dependencies, reducing the final image size. The entrypoint uses `dumb-init`[11] to handle signal forwarding and prevent zombie processes.

The entrypoint script registers a `SIGTERM` handler that calls the service's `destroy()` method, which performs an ordered shutdown: the firehose processor, Constellation processor, and DM notification processor are stopped first, then the HTTP server is terminated using `http-terminator`[12] to allow in-flight requests to complete, and finally the database connection is closed. This ensures that no requests are dropped, and no data is lost during container restarts or rolling deployments.

An installer shell script automates the deployment of Fedisky alongside an existing PDS installation. The script reads the PDS's existing configuration (hostname, admin credentials), prompts the operator for ActivityPub-specific settings (hostname, bridge account preferences), and generates the environment configuration, Caddy reverse proxy rules, Docker Compose service definition, and `systemd` unit file. The Caddy configuration uses path-based routing to direct ActivityPub-specific paths (such as `/users/*`, `/inbox`, and `/.well-known/webfinger`) to the Fedisky container while forwarding all other traffic to the PDS. Requests with an `Accept: application/activity+json` header are also routed to Fedisky, enabling content negotiation on shared paths. A companion updater script handles upgrades by pulling the latest image, backing up the configuration, performing a rolling restart of the Fedisky container, and cleaning up old Docker images. The Docker Compose configuration declares a health dependency between the Fedisky and PDS containers, ensuring that Fedisky only starts after the PDS's health check passes.

[11] `https://github.com/Yelp/dumb-init`
[12] `https://www.npmjs.com/package/http-terminator`

4 Discussion

4.1 Sidecar Architecture

Fedisky is designed as a *sidecar container* as defined in [23, Ch. 3]: a secondary container that provides auxiliary functionality to its *application container*—in this case, a Bluesky PDS—while sharing system resources and running in sync with it [23, p. 21]. Two alternative architectures were considered: a *centralized bridging service*, as realized by Bridgy Fed (Section 2.4.1), which mediates all cross-protocol traffic through a single instance; and a *PDS fork*, which integrates bridging logic directly into the PDS codebase. The following subsections compare these three approaches along the dimensions of user experience, scalability, maintainability, and modularity.

4.1.1 User Experience

A key advantage of the sidecar architecture is that it provides a unified identity for users across both protocols. Because the sidecar shares the PDS's hostname, a Bluesky user with the handle `alice.fedisky.social` is discoverable in the Fediverse as `@alice@fedisky.social`. This is a direct consequence of the sidecar sharing system resources with the application container, and would not be possible with a centralized bridge that operates under its own domain. In effect, this property grants the user a form of *dual network citizenship*: a single account that is simultaneously a first-class participant in the AT Protocol network and the ActivityPub Fediverse, addressable under one consistent handle and on terms set by the operator the user already chose.

As described in Chapter 3, Fedisky includes the user's AT Protocol URI in the ActivityPub actor's `alsoKnownAs` field, following the account links approach proposed by Barrett [4]. This allows Fediverse instances to link the bridged actor back to the original Bluesky profile. However, the linking is currently unidirectional. For full bidirectional linking, the user's DID document would also need to reference the ActivityPub actor URI. Since DID documents for `did:plc` identifiers are managed by the PDS, achieving this would require either a user verification flow via email, or a modified PDS that automatically includes the ActivityPub actor URI when generating DID documents. The former adds friction to the user experience, while the latter reintroduces the tight coupling to PDS internals that the sidecar pattern is designed to avoid.

The bridging is also not fully transparent with respect to authorship. Fediverse replies arrive in Bluesky threads attributed to the Mastodon bridge account rather than the original Fediverse author, prefixed with an attribution line such as "@bob@mastodon.social replied:". This is a fundamental limitation: the bridge account cannot create arbitrary Bluesky accounts for each Fediverse user, so it must relay replies through a shared bridge

account. Similarly, external Bluesky replies discovered via the Constellation API are attributed to the Bluesky bridge account when delivered to the Fediverse. While the attribution prefix preserves authorship information, the indirection reduces the fidelity of conversations compared to native interactions on either platform.

For Fediverse users to appear as native actors within Bluesky—rather than being relayed through bridge accounts—significant changes across the Bluesky stack would be necessary. The AT Protocol identity system would need to accommodate non-AT-Protocol identifiers, either through a new DID method that wraps ActivityPub actor URIs or through a mapping mechanism that assigns DIDs to Fediverse users. The AppView, which indexes content from the AT Protocol relay network, would need to additionally index ActivityPub content and resolve Fediverse profiles into the `app.bsky.actor.profile` schema. Bluesky client applications, which currently assume AT Protocol-native profiles with handles resolvable via DNS, would need to render profiles originating from ActivityPub and handle Fediverse-style addresses such as `@bob@mastodon.social`. These changes would span the PDS, the AppView, the relay network, and client implementations, and are well beyond the scope of a sidecar service.

Engagement visibility is also asymmetric. When a Fediverse user likes or reposts a bridged post, the engagement is stored locally and the Bluesky author is notified via a batched direct message after a configurable delay. This contrasts with native Bluesky notifications, which appear immediately in the user's notifications feed. The batching approach avoids flooding users with individual messages but introduces latency and moves engagement signals from the notification feed to the chat interface, which users may not monitor as closely.

Neither the sidecar nor a PDS fork can address this limitation, because Bluesky's notification system is not controlled by the PDS. Notifications are generated by the AppView, which indexes firehose events and serves them to clients via the `app.bsky.notification.listNotifications` endpoint. Since the AppView is a separate service outside the PDS operator's control, no PDS-level modification can inject Fediverse engagements into the native notification feed. A centralized bridge like Bridgy Fed faces the same constraint, though it can maintain persistent actor representations for Fediverse users, making cross-platform conversations appear more natural on the Fediverse side.

4.1.2 Scalability

The sidecar architecture inherently limits the scope of bridging to a single PDS instance, which simplifies scaling considerations. Each sidecar only processes the firehose of its own PDS and only maintains follower relationships for its local users, keeping resource consumption proportional to the size of the PDS rather than the size of the network.

Within a single PDS, the primary scaling bottleneck is the database. Fedisky uses SQLite, which provides simplicity and eliminates the need for a separate database server, but imposes a single-writer constraint. As the number of ActivityPub followers grows, the volume of inbox deliveries and database writes for follower management, post mapping and engagement tracking increases accordingly. For small to medium PDS instances, this is unlikely to be a concern, but large instances with many active users could encounter

write contention.

A second consideration is outbound delivery fan-out. When a Bluesky user with many Fediverse followers publishes a post, the sidecar must deliver the corresponding `Create` activity to each follower's inbox. Fedify handles delivery asynchronously, and the shared inbox optimization—where a single delivery serves all followers on the same Fediverse instance—reduces the number of HTTP requests. Nevertheless, users with followers spread across many Fediverse instances will generate significant outbound traffic.

The Constellation polling mechanism introduces a third scaling factor. Each monitored post requires an API call to the Constellation service, and the processor fetches up to 50 posts per polling interval. As the number of active posts grows, some may experience delayed external reply discovery. This is an inherent trade-off of the polling approach: real-time discovery would require either a push-based notification from Constellation or a firehose subscription to the broader AT Protocol network, both of which would significantly increase complexity and resource requirements.

By contrast, a centralized bridge amortizes certain costs across all users, but must handle proportionally more traffic. A PDS fork could potentially optimize database access by sharing the PDS's existing storage layer, avoiding the overhead of a separate database. The sidecar trades these potential optimizations for deployment simplicity and independence from PDS internals.

4.1.3 Maintainability

A primary motivation for choosing the sidecar pattern was to minimize the maintenance burden relative to a PDS fork. Because Fedisky communicates with the PDS exclusively through its XRPC API whose endpoints are defined by AT Protocol Lexicons, it is insulated from internal PDS changes. As long as the PDS continues to expose the same API surface, Fedisky remains compatible regardless of internal refactoring, dependency updates, or implementation changes in the PDS.

This decoupling has practical consequences. The PDS reference implementation is actively developed, with frequent releases that modify internal data structures, database schemas, and processing pipelines. A fork would need to continuously rebase on these changes, resolving merge conflicts between upstream updates and bridging modifications. The sidecar avoids this entirely: updates to the PDS and updates to Fedisky are independent operations that can be performed on different schedules.

The sidecar's own maintenance surface is also contained. The conversion layer is organized around a converter registry that maps AT Protocol Lexicon collections to converter implementations. Adding support for a new record type requires implementing a new converter and registering it, without modifying existing converters or core infrastructure. This extensibility is important as both protocols continue to evolve: new Lexicon types or ActivityPub activity types can be accommodated incrementally.

However, the sidecar introduces its own maintenance obligations. It depends on the Fedify library for ActivityPub federation, the `@atproto` packages for AT Protocol interaction, and the Constellation API for external reply discovery. Changes to any of these dependencies require updates to the sidecar. In particular, the Constellation API is

not yet standardized, and changes to its interface or availability would directly impact external reply discovery.

4.1.4 Modularity

The sidecar architecture enforces a clear module boundary between the PDS and the bridging logic. All communication crosses this boundary via HTTP, which means the two components can be developed, tested, deployed, and scaled independently. This separation also enables technology diversity: Fedisky is implemented in TypeScript, while the PDS reference implementation is written in TypeScript as well, but any AT Protocol-compliant PDS regardless of implementation language could use Fedisky as its sidecar.

Within Fedisky itself, modularity is reflected in the separation between processors, converters, and federation endpoints. The `FirehoseProcessor`, `ConstellationProcessor`, and `DmNotificationProcessor` operate independently, each with its own lifecycle and scheduling. The converter registry decouples record type handling from the processing pipeline: processors delegate conversion to the registry without knowledge of specific record types. Federation endpoints handle inbound ActivityPub activities through a dispatcher that routes by activity type, again without coupling to specific conversion logic.

This internal modularity has practical benefits for testing. Converters can be unit-tested in isolation with mock inputs, processors can be tested against recorded firehose frames, and federation endpoints can be tested with mocked HTTP requests. The end-to-end test suite orchestrates a full deployment with a real PDS, mock Fediverse instance, and mock Constellation service, verifying that the components integrate correctly.

A PDS fork would sacrifice much of this modularity. Bridging logic interleaved with PDS code would be harder to test in isolation, and changes to one concern could inadvertently affect the other. A centralized bridge maintains modularity at the service level but must handle a broader set of concerns like user management and abuse prevention, that the sidecar delegates to the PDS operator.

4.1.5 Summary

Table 4.1 summarizes the comparison of the three architectural approaches across the four dimensions discussed above.

4.2 Challenges and Trade-offs in Protocol Bridging

Beyond architectural considerations, bridging AT Protocol and ActivityPub presents several protocol-level challenges. This section reflects on the most significant ones encountered during the implementation of Fedisky, analyzing the design decisions made and their implications.

Dimension	**Sidecar (Fedisky)**	**Centralized Bridge**	**PDS Fork**
User Experience	Unified identity via shared domain; authorship attribution via bridge accounts	Separate domain for bridged identities; persistent per-user actors on the Fediverse side	Unified identity; could access PDS internals for deeper integration
Scalability	Proportional to single PDS; SQLite single-writer constraint; shared inbox optimization	Must handle network-wide traffic; amortizes infrastructure costs across all users	Shares PDS storage layer; no separate database overhead
Maintainability	Decoupled via stable XRPC API; independent update schedule; depends on Fedify and Constellation	Similar dependencies; can invest more resources given broader user base	Tightly coupled to PDS internals; continuous rebasing on upstream changes
Modularity	Clear HTTP boundary; independent testing and deployment; technology-agnostic	Service-level modularity; broader concerns (abuse prevention, user management)	Interleaved concerns; harder to test in isolation; locked to PDS technology stack

Table 4.1: Comparison of architectural approaches for cross-protocol bridging.

4.2.1 Identity Projection

Fedisky derives each user's ActivityPub actor URI from their DID, producing a stable, unidirectional identity projection (Chapter 3). This design decision has two notable consequences worth analyzing.

First, the projection is deliberately unidirectional: Bluesky users gain ActivityPub actors, but Fediverse users do not gain AT Protocol identities. An alternative would be to create AT Protocol accounts for each interacting Fediverse user, which would enable native-looking interactions on the Bluesky side. However, this would raise questions of consent (creating accounts without explicit user action), resource consumption (each account requires a repository, signing keys, and DID registration), and identity ownership (who controls the DID?). Bridgy Fed takes a version of this approach, maintaining persistent proxy identities for bridged users, but it operates as a centralized service with the resources and governance structure to manage these identities at scale. For a per-instance sidecar, the simpler unidirectional projection is a pragmatic choice that avoids these complexities at the cost of asymmetric authorship attribution.

Second, the use of the DID rather than the Bluesky handle as the canonical anchor ensures stability across handle changes, but it produces actor URIs that are opaque to humans (`/users/did:plc:...` rather than `/users/alice`). This is a deliberate trade-off: handle-based URIs would be more readable but would break if the user changes their handle, invalidating all existing follow relationships in the Fediverse. The DID-based approach prioritizes correctness over aesthetics.

4.2.2 Content Format Conversion

The conversion between AT Protocol's plain-text-with-facets representation and ActivityPub's HTML is inherently lossy in both directions (Section 3.5). The key question is whether a higher-fidelity conversion is achievable, and the answer depends on which direction is considered.

In the outbound direction (AT Protocol to ActivityPub), the conversion is nearly lossless: AT Protocol's facet types (links, mentions, hashtags) all have direct HTML equivalents, and paragraph structure is preserved through double-newline splitting. The main limitation is that AT Protocol's content warning system (self-labels with machine-readable identifiers like `porn` or `nudity`) must be mapped to ActivityPub's free-text `summary` field, which conflates the label's semantic meaning with its display text.

In the inbound direction (ActivityPub to AT Protocol), the loss is more significant. Mastodon and other Fediverse implementations support rich HTML including bold, italic, lists, block quotes, and code blocks—none of which have AT Protocol facet equivalents. Fedisky extracts plain text and preserves only links and mentions. This is not a limitation of the bridge implementation but of AT Protocol's content model: until the protocol introduces facet types for additional formatting, any bridge must discard this information. The Bluesky community has discussed richer text formatting [34], but no specification has been adopted as of this writing.

4.2.3 Delivery Mechanism Bridging

Translating between AT Protocol's pull-based firehose and ActivityPub's push-based inbox delivery is architecturally straightforward (Chapter 3), but introduces two asymmetries worth examining.

The first is fan-out cost. A single firehose event for a popular user may require delivery to hundreds of Fediverse inboxes. ActivityPub's shared inbox optimization mitigates this when multiple followers reside on the same instance, but the sidecar must still handle significant outbound traffic for popular users. This is an inherent cost of the push-based model that any bridge must bear; even Mastodon's native federation faces the same fan-out challenge.

The second asymmetry concerns external reply discovery. Replies from users on other Bluesky PDS instances do not appear in the local firehose, requiring Fedisky to poll the Constellation backlinks API. This creates a two-tier latency model: local interactions are federated in real time, while external replies may be delayed by up to one polling interval (default 60 seconds). A real-time alternative would be subscribing to the network-wide relay firehose and filtering for relevant replies, but this would require processing the entire AT Protocol network's traffic—a disproportionate resource cost for a per-instance sidecar. A centralized bridge like Bridgy Fed can justify this cost because it serves the entire network.

4.2.4 Engagement Semantics

The mismatch between AT Protocol's persistent engagement records and ActivityPub's transient engagement activities creates an asymmetry that no bridge architecture can fully resolve without changes to Bluesky's infrastructure.

The core issue is that Bluesky's notification system is controlled by the AppView, not the PDS. The AppView indexes likes and reposts from the AT Protocol relay network and serves them to clients via the `app.bsky.notification.listNotifications` endpoint. Since the AppView is outside the PDS operator's control, no PDS-level modification—whether sidecar, fork, or centralized bridge—can inject Fediverse engagements into the native notification feed. Fedisky's workaround of batching engagements into direct messages is a pragmatic solution that preserves the information but changes its presentation and timing.

In the outbound direction, the mapping is more natural: AT Protocol likes and reposts map directly to ActivityPub `Like` and `Announce` activities. This directional asymmetry—outbound engagement mapping is straightforward while inbound engagement is constrained—reflects a broader pattern in the bridge: AT Protocol's separation of concerns (PDS stores data, AppView indexes it, clients display it) means that a sidecar operating at the PDS level can only influence the data storage layer, not the presentation layer.

4.2.5 Bridging Loop Prevention

Because the sidecar creates posts on the PDS through bridge accounts and simultaneously consumes the PDS's firehose, there is a risk of infinite bridging loops. Fedisky prevents this by filtering bridge account DIDs from both the firehose processor and the Constellation processor. This is a simple but essential safeguard. The simplicity of the solution is itself worth noting: more complex bridging topologies (e.g., multiple bridges operating on the same PDS, or chains of bridges across protocols) would require more sophisticated deduplication mechanisms, such as tagging bridged content with provenance metadata that downstream bridges could inspect.

4.3 Network-Level Implications

Beyond the per-component trade-offs discussed above, Fedisky inherits a more fundamental dependency through its identity model: every actor URI it produces is anchored in a `did:plc` identifier, and the validity of every signature it serves to the Fediverse depends on the PLC directory remaining available, honest, and stable in its current shape. As described in Section 2.1.2, the directory is operated today by a single entity and its long-term governance is unsettled. The default rotation-key custody pattern compounds this concern: a Fedisky-bridged actor whose rotation keys sit entirely with its PDS operator is, in principle, redirectable or impersonable by that operator without any signal visible to its Fediverse followers, since the actor's signing material would continue to validate against a re-keyed DID document. Two practical implications follow. First, operators deploying Fedisky should encourage users to enroll self-controlled rotation keys, moving authoritative custody out of the PDS. Second, the bridge ecosystem benefits from the same end-state that Bluesky's decentralization roadmap aims for: many independent PDSs, with key custody distributed across them rather than concentrated in a small number of large operators. The remainder of this section examines what such a shift would mean at the network level.

If Fedisky-style per-instance bridging gains traction, the cost calculus for new Fediverse operators changes. Operating a full Mastodon stack involves a Rails web server, background workers, a media store, a federation queue, and the moderation tooling expected of a modern Fediverse instance. A PDS plus sidecar, by contrast, is a much smaller operational surface: a self-contained AT Protocol server, a stateless converter, and a SQLite database. From the Fediverse's outside view, the two are largely indistinguishable. Both expose WebFinger, both deliver `Create` activities to peer inboxes, both serve actor profiles. It is therefore plausible that, over time, some fraction of new operators choose PDS plus Fedisky over Mastodon, and some existing Mastodon administrators facing operational strain migrate. The Fediverse would continue to look the same from outside, but its internal infrastructure would shift.

This shift introduces a structural change. A native Mastodon instance has only two dependencies for participating in the network: its own server and the HTTPS-reachable inboxes of the peers it federates with. A Fedisky-backed instance additionally depends

on shared AT Protocol infrastructure: the PLC directory, the Constellation backlinks API, the AppView, and, more loosely, the broader relay network. Each of these is, today, a centralized or single-vendor component. The Fediverse a Fedisky operator joins is therefore a Fediverse anchored to AT Protocol infrastructure that the operator does not control.

The asymmetry surfaces in failure modes. A Mastodon instance going dark removes one node from the network; the rest is unaffected. An outage or compromise of a shared AT Protocol component, by contrast, would affect every Fedisky-backed node simultaneously. If the PLC directory is unreachable, the actor URIs and signing keys of every Fedisky-bridged user across every PDS become unverifiable to remote Mastodon servers at the same moment. If the Constellation backlinks API degrades, reply threading silently breaks across all Fedisky deployments. A future contentious change to PLC governance—to which the protocol team itself concedes the system is still on a journey [18]—would, in the same way, implicate every Fedisky deployment at once. The danger is not that any single such failure is likely in the short term, but that a Fediverse heavily built on Fedisky-style bridges would, for the first time, exhibit correlated failure modes across nodes that today's Fediverse does not.

These risks are manageable, and several mitigations align and reinforce work already underway. Most directly, distributing identity authority is the highest-leverage intervention: the planned independent governance and federated mirroring of the PLC directory, together with operator and user practices that move authoritative rotation keys out of any single PDS, removes the largest shared dependency from the critical path. Second, Fedisky itself can maintain a local mirror of the PLC records (i.e. DID documents) it has already encountered, so that known counter-parties remain verifiable during a directory outage. For a small-to-medium PDS, this amounts to a few thousand records and well under 100 MB of storage. Third, the bridge should fail open rather than closed: when an AT-side dependency is unavailable, Fedisky can continue accepting inbound deliveries and serving cached actor data, allowing local-to-Fediverse interactions to continue uninterrupted. Fourth, single-vendor dependencies such as the Constellation API benefit from provider plurality or, eventually, a standardized cross-PDS backlinks discovery mechanism. Finally, the per-instance design itself contributes: many small independent PDSs spread the consequences of any individual operator's failure or misbehavior thinly across the network, in contrast to a few large operators whose failure would be concentrated.

The framing matters; the point is not that Fedisky-style bridging is harmful to the Fediverse, but that a federation built increasingly on a shared infrastructure inherits that infrastructure's failure modes. Recognizing this early and designing both the bridge and the surrounding ecosystem defensively against it is what allows decentralized cross-platform federation to remain genuinely decentralized rather than only appearing so.

4.4 Limitations and Future Work

Several limitations of the current implementation suggest directions for future work.

Feature Coverage. Fedisky currently bridges posts, replies, likes, and follows. It does not support direct messages, polls, lists, custom feeds, or account migration. Extending coverage to additional record types is facilitated by the converter registry pattern, but each new type requires careful analysis of semantic differences between the protocols.

Bidirectional Following. The current implementation supports Fediverse users following Bluesky users, but not the reverse. A Bluesky user cannot follow a Fediverse account through the bridge. Supporting this would require the sidecar to maintain ActivityPub actor representations for remote Fediverse users and subscribe to their outboxes or accept forwarded activities, adding significant complexity.

Database Scalability. SQLite's single-writer constraint limits concurrent write throughput. For PDS instances with a large number of users or high activity volumes, migrating to a client-server database such as PostgreSQL would improve write concurrency at the cost of additional deployment complexity.

Real-time External Replies. The polling-based approach to external reply discovery introduces latency that could be reduced by subscribing to the broader AT Protocol relay network. However, this would substantially increase bandwidth and processing requirements, as the sidecar would need to filter a network-wide firehose for relevant replies.

Standardization. The Constellation backlinks API, which Fedisky relies on for external reply discovery, is not yet standardized. Changes to this API or its deprecation would require Fedisky to adopt an alternative discovery mechanism.

Consent. The current implementation bridges all posts from users on the PDS without requiring explicit opt-in. While this maximizes coverage, it raises questions about user consent and control over cross-platform visibility. A future version could introduce per-user opt-in or opt-out mechanisms, allowing users to control whether their content is federated to the Fediverse.

5 Conclusion

This thesis presented Fedisky, a sidecar service that enables ActivityPub federation for an AT Protocol Personal Data Server. Fedisky allows Fediverse users to discover, follow, and interact with Bluesky users on a PDS without requiring any modifications to the PDS itself.

RQ1 asked how a sidecar service can enable ActivityPub federation for an AT Protocol PDS. Fedisky answers this by consuming the PDS's repository commit firehose, converting AT Protocol records into ActivityPub activities, and delivering them to followers' inboxes. In the reverse direction, it receives ActivityPub activities at standard federation endpoints, converts them to AT Protocol records, and writes them to the PDS via its XRPC API. Identity is bridged by deriving each user's ActivityPub actor URI from their DID, and the sidecar's co-location with the PDS enables a unified identity where a Bluesky handle like `alice.fedisky.social` becomes `@alice@fedisky.social` in the Fediverse—a form of *dual network citizenship* under a single, operator-chosen handle. External replies from other Bluesky PDS instances are discovered through polling the Constellation backlinks API and federated to followers.

RQ2 asked about the key technical challenges and trade-offs. We identified four primary areas. Identity projection is unidirectional: Bluesky users are automatically represented as ActivityPub actors, but Fediverse users do not gain AT Protocol identities, and inbound interactions must be relayed through a bridge account. Content format conversion between AT Protocol's byte-range facets and ActivityPub's HTML is inherently lossy, as rich HTML formatting has no direct representation in the facet system. Delivery mechanism bridging translates between AT Protocol's pull-based firehose and ActivityPub's push-based inbox delivery, introducing fan-out costs for outbound federation and polling latency for external reply discovery. Engagement notifications cannot be injected into Bluesky's native notification feed, as this is controlled by the AppView rather than the PDS, requiring a workaround via batched direct messages.

RQ3 asked how the sidecar architecture compares to alternative approaches. We compared it against a centralized bridging service (as realized by Bridgy Fed) and a PDS fork along the dimensions of user experience, scalability, maintainability, and modularity. The sidecar provides a unified identity through domain sharing, which a centralized bridge cannot offer. It avoids the maintenance burden of tracking upstream PDS changes that a fork would impose, communicating only through stable API contracts defined by AT Protocol Lexicons. Its scope is limited to a single PDS instance, which simplifies scaling but means each operator must deploy independently. The sidecar enforces a clear module boundary via HTTP, enabling independent development, testing, and deployment, while a fork would tightly couple bridging logic with PDS internals. The trade-off is that the sidecar cannot access these PDS internals for deeper integrations.

In conclusion, the implementation demonstrated that per-instance, decentralized bridging between AT Protocol and ActivityPub is feasible using the sidecar pattern. The converter registry design allows incremental extension to new record types as both protocols evolve. However, the current implementation is limited to posts, replies, likes, and follows, and the bridging remains asymmetric in several respects, most notably in authorship attribution and engagement visibility.

Bibliography

[1] Kevin Åberg Kultalahti. *Who Actually Owns Your ATProto Identity? Hint: It's Probably Not You*. Mar. 1, 2026. URL: https://kevinak.se/blog/who-actually-owns-your-atproto-identity-hint-its-probably-not-you (visited on Apr. 16, 2026).

[2] Gabriel Amador. *wafrn/wafrn: Tumblr-inspired Federated Social Network*. URL: https://codeberg.org/wafrn/wafrn (visited on Apr. 16, 2026).

[3] Alex Auvolat and François Taïani. "Merkle Search Trees: Efficient State-Based CRDTs in Open Networks". In: *38th IEEE International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, Oct. 2019, pp. 221–230. DOI: 10.1109/SRDS.2019.00032.

[4] Ryan Barrett. *Bridging Identity with Account Links*. A New Social. Aug. 13, 2025. URL: https://blog.anew.social/bridging-identity-with-account-links/ (visited on Apr. 16, 2026).

[5] Ryan Barrett. *Bridgy Fed – Design*. URL: https://bridgy-fed.readthedocs.io/source/design.html (visited on Apr. 16, 2026).

[6] Ryan Barrett. *Possible Futures for Bridgy Fed*. Nov. 1, 2024. URL: https://snarfed.org/2024-11-01_53932 (visited on Apr. 16, 2026).

[7] Ryan Barrett. *Re-Introducing Bridgy Fed*. Nov. 27, 2023. URL: https://snarfed.org/2023-11-27_re-introducing-bridgy-fed (visited on Apr. 16, 2026).

[8] Ryan Barrett and Anuj Ahooja. *Bridgy Fed Documentation*. URL: https://fed.brid.gy/docs (visited on Apr. 16, 2026).

[9] Bluesky Social PBC. *2024 in Review*. Dec. 30, 2024. URL: https://bsky.social/about/blog/12-30-2024-year-in-review (visited on Apr. 16, 2026).

[10] Bluesky Social PBC. *Creating an Independent Public Ledger of Credentials (PLC) Directory Organization*. Sept. 19, 2025. URL: https://docs.bsky.app/blog/plc-directory-org (visited on Apr. 16, 2026).

[11] Bluesky Social PBC. *Data Repositories*. URL: https://atproto.com/specs/repository (visited on Apr. 16, 2026).

[12] Bluesky Social PBC. *DID Identity*. URL: https://atproto.com/specs/did (visited on Apr. 16, 2026).

[13] Bluesky Social PBC. *did:plc – DID Method Specification v0.3.0*. Specification. Dec. 2025. URL: https://web.plc.directory/spec/v0.1/did-plc (visited on Apr. 16, 2026).

[14] Bluesky Social PBC. *Event Streams*. URL: https://atproto.com/specs/event-stream (visited on Apr. 16, 2026).

[15] Bluesky Social PBC. *Handle*. URL: https://atproto.com/specs/handle (visited on Apr. 16, 2026).

[16] Bluesky Social PBC. *Labels and Moderation | Bluesky*. URL: https://docs.bsky.app/docs/advanced-guides/moderation (visited on Apr. 16, 2026).

[17] Bluesky Social PBC. *Lexicon*. URL: https://atproto.com/specs/lexicon (visited on Apr. 16, 2026).

[18] Bluesky Social PBC. *Protocol Check-in (Fall 2025)*. Oct. 20, 2025. URL: https://docs.bsky.app/blog/protocol-checkin-fall-2025 (visited on Apr. 16, 2026).

[19] Bluesky Social PBC and Contributors. *app.bsky.embed.images Lexicon*. URL: https://github.com/bluesky-social/atproto/blob/main/lexicons/app/bsky/embed/images.json (visited on Apr. 16, 2026).

[20] Bluesky Social PBC and Contributors. *app.bsky.embed.video Lexicon*. URL: https://github.com/bluesky-social/atproto/blob/main/lexicons/app/bsky/embed/video.json (visited on Apr. 16, 2026).

[21] Bluesky Social PBC and Contributors. *app.bsky.feed.post Lexicon*. URL: https://github.com/bluesky-social/atproto/blob/main/lexicons/app/bsky/feed/post.json (visited on Apr. 16, 2026).

[22] Ivan Burmistrov. *All You Need Is Wide Events, Not "Metrics, Logs and Traces"*. A Song Of Bugs and Patches. Feb. 15, 2024. URL: https://isburmistrov.substack.com/p/all-you-need-is-wide-events-not-metrics (visited on Apr. 16, 2026).

[23] Brendan Burns. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services, Using Kubernetes*. Second Edition. Sebastopol: O'Reilly, 2024. 200 pp. ISBN: 978-1-0981-5635-0.

[24] Clare Duffy. *Bluesky's User Base Has Doubled in the Past 90 Days. Is It a Mass Exodus from X? | CNN Business*. CNN. Nov. 13, 2024. URL: https://www.cnn.com/2024/11/13/tech/x-musk-bluesky-users-post-election (visited on Apr. 16, 2026).

[25] Jiahui He et al. "Flocking to Mastodon: Tracking the Great Twitter Migration". In: *Proceedings of the 2023 ACM on Internet Measurement Conference*. ACM, 2023, pp. 111–123. DOI: 10.1145/3618257.3624819.

[26] Paul Jones et al. *WebFinger*. RFC 7033. IETF, Sept. 2013. URL: https://datatracker.ietf.org/doc/html/rfc7033 (visited on Apr. 16, 2026).

[27] Martin Kleppmann et al. "Bluesky and the AT Protocol: Usable Decentralized Social Media". In: *Proceedings of the ACM CoNEXT-2024 Workshop on the Decentralization of the Internet*. DIN '24. Los Angeles, CA, USA: ACM, Dec. 9, 2024. ISBN: 979-8-4007-1252-4. DOI: 10.1145/3694809.3700740.

[28] Lucio La Cava, Emilio Ferrara, and Sergio Ferretti. "Drivers of Social Influence in the Twitter Migration to Mastodon". In: *Scientific Reports* 13.1 (2023), p. 21626. DOI: 10.1038/s41598-023-48200-7.

[29] Charity Majors, Liz Fong-Jones, and George Miranda. *Observability Engineering: Achieving Production Excellence*. First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2022. 295 pp. ISBN: 978-1-4920-7644-5.

[30] Mastodon GmbH. *ActivityPub - Mastodon Documentation*. Nov. 21, 2025. URL: https://docs.joinmastodon.org/spec/activitypub/ (visited on Apr. 16, 2026).

[31] Renato Mendes. "Deep Dive: Wafrn's Dual-Protocol Federation". In: (Feb. 26, 2026). URL: https://rmendes.net/articles/2026/02/26/deep-dive-wafrn-dual-protocol-federation/ (visited on Apr. 16, 2026).

[32] Bryan Newbold and Daniel Holmgren. *Authenticated Transfer: Architecture Overview*. Internet-Draft. IETF, Sept. 14, 2025. URL: https://datatracker.ietf.org/doc/draft-newbold-at-architecture/ (visited on Apr. 16, 2026).

[33] Michael Prorock and Oliver Terbu. *did:web Method Specification*. URL: https://w3c-ccg.github.io/did-method-web/ (visited on Apr. 16, 2026).

[34] SaphireLattice. *Formatting and Rich Text*. GitHub. Dec. 6, 2024. URL: https://github.com/bluesky-social/social-app/issues/6998 (visited on Apr. 16, 2026).

[35] James Snell and Evan Prodromou. *Activity Streams 2.0*. W3C Recommendation. W3C, May 23, 2017. URL: https://www.w3.org/TR/2017/REC-activitystreams-core-20170523/ (visited on Apr. 16, 2026).

[36] James Snell and Evan Prodromou. *Activity Vocabulary*. W3C Recommendation. W3C, May 2017. URL: https://www.w3.org/TR/2017/REC-activitystreams-vocabulary-20170523/ (visited on Apr. 16, 2026).

[37] Manu Sporny, Gregg Kellogg, and Markus Lanthaler. *JSON-LD 1.0*. W3C Recommendation. W3C, Jan. 16, 2014. URL: https://www.w3.org/TR/2014/REC-json-ld-20140116/ (visited on Apr. 16, 2026).

[38] Manu Sporny and Dmitri Zagidulin. *Decentralized Identifiers (DIDs) v1.1*. W3C Candidate Recommendation. W3C, Mar. 5, 2026. URL: https://www.w3.org/TR/2026/CR-did-1.1-20260305/ (visited on Apr. 16, 2026).

[39] Manu Sporny et al. *Decentralized Identifiers (DIDs) v1.0*. W3C Recommendation. W3C, July 19, 2022. URL: https://www.w3.org/TR/2022/REC-did-core-20220719/ (visited on Apr. 16, 2026).

[40] Jessica Tallon and Christine Lemmer-Webber. *ActivityPub*. W3C Recommendation. W3C, Jan. 2018. URL: https://www.w3.org/TR/2018/REC-activitypub-20180123/ (visited on Apr. 16, 2026).

[41] *Wafrn – FAQ and User Guide*. URL: https://wafrn.net/faq/user.html (visited on Apr. 16, 2026).

Acronyms

CAR Content Addressable aRchive. 11

CBOR Concise Binary Object Representation. 10, 11, 22, 27, 34

CID Content Identifier. 31, 34

DID Decentralized Identifier. 2, 3, 5–11, 16, 22, 24, 26, 27, 29, 31, 35, 39, 40, 44, 46, 47, 49

DNS Domain Name System. 2, 7, 9, 40

HTML Hypertext Markup Language. 2, 11, 14, 15, 24, 27, 31, 33, 34, 44, 49

HTTP Hypertext Transfer Protocol. 8, 10, 13, 15, 16, 20, 21, 23, 27, 29, 31, 32, 35–38, 41–43, 49

HTTPS Hypertext Transfer Protocol Secure. 2, 7, 9, 13, 14, 21, 35, 37, 46

JRD JSON Resource Descriptor. 14

JSON JavaScript Object Notation. 10, 12, 13, 16

MST Merkle Search Tree. 10

NSID Namespace Identifier. 10, 11

PDS Personal Data Server. iii, v, 2, 3, 6–9, 11, 12, 16, 17, 19–24, 26, 27, 29, 31–35, 37–43, 45–49

PLC Public Ledger of Credentials. 6, 46, 47

SSRF Server-Side Request Forgery. 21, 35

TID Timestamp Identifier. 33, 34

TLS Transport Layer Security. 7, 21, 37

URI Uniform Resource Identifier. 2, 3, 5–7, 10–14, 17, 23, 26, 27, 29, 31–34, 39, 40, 44, 46, 47, 49

URL Uniform Resource Locator. 7, 9, 14, 16, 21, 23, 24, 26, 27, 29, 31, 33–35

XSS Cross-Site Scripting. 24, 33

A Appendix

A.1 Fedisky Environment Variables

Table A.1 lists all environment variables supported by Fedisky. Variables marked as required must be set for the service to start; all others have default values as specified.

Variable	Type	Default
PDS_URL	String	**Required**
PDS_HOSTNAME	String	**Required**
PDS_ADMIN_TOKEN	String	**Required**
AP_PORT	Integer	2588
AP_HOSTNAME	String	localhost
AP_PUBLIC_URL	String	Derived from hostname
AP_VERSION	String	0.0.0
AP_DB_LOCATION	String	:memory:
AP_FIREHOSE_ENABLED	Boolean	true
AP_FIREHOSE_CURSOR	String	–
AP_MASTODON_BRIDGE_ENABLED	Boolean	true
AP_MASTODON_BRIDGE_HANDLE	String	mastodon.{hostname}
AP_MASTODON_BRIDGE_DISPLAY_NAME	String	Mastodon Bridge
AP_MASTODON_BRIDGE_DESCRIPTION	String	See source
AP_MASTODON_BRIDGE_AVATAR_URL	String	Mastodon logo URL
AP_BLUESKY_BRIDGE_ENABLED	Boolean	true
AP_BLUESKY_BRIDGE_HANDLE	String	bluesky.{hostname}
AP_BLUESKY_BRIDGE_DISPLAY_NAME	String	Bluesky Bridge
AP_BLUESKY_BRIDGE_DESCRIPTION	String	See source
AP_BLUESKY_BRIDGE_AVATAR_URL	String	Bluesky logo URL
AP_CONSTELLATION_URL	String	Empty (disabled)
AP_CONSTELLATION_POLL_INTERVAL	Integer	60000
AP_APPVIEW_URL	String	public.api.bsky.app
AP_DM_NOTIFICATIONS_ENABLED	Boolean	true
AP_DM_NOTIFICATIONS_POLL_INTERVAL	Integer	300000
AP_DM_NOTIFICATIONS_BATCH_DELAY	Integer	600000
AP_ALLOW_PRIVATE_ADDRESS	Boolean	false

Table A.1: Fedisky environment variables.

A.2 Documentation of AI Assistance Tools

Tool	Usage
DeepL	Translation of the abstract into German. Reviewed and edited manually.
Anthropic Claude Code Opus 4.6	Code review, refactoring, and implementation suggestions. All suggestions were reviewed, adapted, and tested.
Anthropic Claude Sonnet 4.6	Used for brainstorming and outlining the overall thesis structure, summarizing related work, and proofreading for clarity and grammar. Suggestions were reviewed and adapted manually.

Table A.2: Documentation of AI Assistance Tools.